Studying the characterisation of deep CNN neurons

Computer Science Department

July 2^{nd}, 2019

Author: Carles Garriga Estradé

Supervisor: Dario Garcia Gasulla
Co-supervisor: Javier Bejar Alonso

MASTER IN INNOVATION AND RESEARCH IN INFORMATICS
Facultat d'informàtica de Barcelona
UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC) – BarcelonaTech

Abstract

This master thesis has studied the characteristics of deep conventional neurons present in a pre-trained Deep Convolutional Neural Network structure based on the GoogleNet network. Three approaches towards finding different behaviours in the architecture have been introduced: supervised approach to find patterns in the neuron weights, an unsupervised kernel clustering method alongside dimensionality reduction for visualisation and Topological Data Analysis on the layer channels.

Using the supervised method, we have found a correlation between neurons' weights and the patterns found in images that excite them most.

To understand the neurons behaviour in each layer of the network, Spectral Clustering has been applied in order to group the neurons by their weights. In conjunction with clustering, a dimensional reduction has been performed using UMAP in order to visualise the data in an embedded 2-Dimensional space. As a result, we have found several groups of neurons for each layer. In some cases, when visualising this embedding, a clear circular shape has been found.

As a final step, in order to get the insights of the actual layers, a topological data analysis has been performed on the channels of each layer. We have found a ring-shaped graph representing the neurons' channels present in a layer, which confirms the idea of CNN layers trying to mimic the primary visual cortex.

Acknowledgements

I have to thank my supervisors, Dario and Javier, for providing assistance during this process. To my family and colleagues, for their comprehension during all this period. To my university friends: Carla, Marçal, Dani and specially to Laura, for all the help and support. To all of them I want to express my wholehearted gratitude.

Contents

List of Figures

List of Algorithms

1 Introduction

Artificial Neural Networks are one of the current methods in Machine Learning that are inspired in real biological neurons, forming a network by connecting them that learns from the inputs and which performs a task of classification or prediction.

Thanks to the invention of *backpropagation* and the computational advances made in the past years and the resources available, such as GPU's and lately TPU's [10], the usage and viability of Artificial Neural Networks has increased a lot.

These networks are currently used in a wide range of areas, such as Natural Language Processing, Sentiment Analysis, Self-driving cars and Speech Recognition, among others.

The first known implementation of a Convolutional Neural Network (CNN) appeared in the '90s [1] as a type of neural network specialised in Image Classification and Computer Vision.

Those CNN were built on a set of connected layers which, in turn, are composed by units that perform similar functions to those of actual neurons. Therefore, the general idea of their structure is a resemblance of an actual visual cortex.

From the very beginning, neural networks have been able to stand against more traditional algorithms and outperform them in the obtained results, thanks to their versatility and the amount of complex information that they can learn. However, complexity in machine learning has its benefits and drawbacks. Because of this, many data scientists and machine learning practitioners are still sceptical about using deep learning methods, considering them *black boxes* due to the amount of complexity and the lack of direct interpretability of such models.

Nevertheless, a big effort towards regaining trust in those methods is being done by trying to provide explainable artificial intelligence, also known as AI. Being part of AI, CNN practitioners have been trying to understand and uncover the insights of the decision-making process that CNN's apply in order to produce an answer or prediction given an input. In order to capture the state of a CNN, representations could be built by extracting the output of any neuron at any layer at a given moment while an input is fed. Thus, resulting in a representation of what each unit of the network is keeping from the actual input in a given moment.

No approaches have been done to extract and group the neurons depending on their actual behaviour or characteristics. We deem this knowledge as equally important, for knowing whether neurons can be grouped by their behaviours would be greatly beneficial computationally-wise. This information could be used to assign predefined and known behaviours to neurons instead of training a CNN from scratch, consequently avoiding the time and hardware resources spent on doing so.

Therefore, the aim of this project is to, given a CNN architecture, extract valuable insights from its neurons different behaviours, as well as try to group them and obtain a more general vision of each group. This study will be done a non-semantical way, in the

[10] Which stands for General (and Tensor) Purpose Units

sense that the output classes will not be used to extract the behaviour of the different units in the architecture.

1.1 Objectives

Having identified the main goal for this thesis, different objectives are set in order to materialise it:

- Identify non-semantic patterns that can be found in units of a given architecture. Those should be interpretable, in order to be able to validate them visually.
- Find the characteristics of those patterns that a given CNN architecture can provide, only by means of the neurons weights.
- Provide tools to classify the previous patterns.
- Use different unsupervised learning techniques to find different groups that share similar characteristics. Those methods will be also applied to discover the bigger picture of the given architecture.

1.2 Introducing a Convolutional Neural Network (CNN)

Since the subject of this thesis is clearly related to CNN, a brief introduction of them and their basic components is necessary.

Input layer

The first layer of the network, also known as the input layer, corresponds to an N-dimensional array of input data.

In the case of 2-dimensional CNNs, this input corresponds to a 3D volume, with the depth dimension corresponding to different channels of the data. For images, this often means three RGB channels, but for other domains this may vary. For the remaining of the document we will stick to RGB inputs. In other words, images, that will be fed to the network by batches of a given size.

Convolutional layer

Convolutional units move (or convolve, as their name suggests) through the entire input image space, applying a element-wise multiplication between a certain part of the image pixels and a matrix of its own (a.k.a kernel or filter) and its posterior summation, resulting as a single output. A basic example can be seen in figure 1.

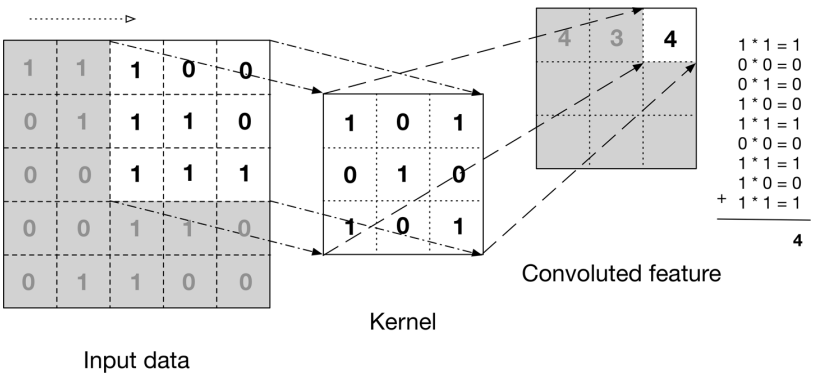


Figure 1: Convolution step with a given input channel and a kernel [11]

This kernel represents the importance given to each pixel of each part the input image is divided in. Bigger the kernel size, higher the number of input pixels that are going to be taken into account for a single output value. Given the amount of kernels found in the convolutional layer, the number of parameters is defined as

$$\text{Num. Params.} = \text{kernel size} \times \text{kernel size} \times \text{input channels} \times \text{output units}$$

. Therefore it can be said that in a CNN, those layers have the majority of trainable parameters.

The number of pixels that a given kernel is allowed to move in order to perform the next convolution is controlled by the stride. Also, both the initial position for the kernel to start and the output shape can be controlled by adding padding to the input. Figure 2, shows an example of a stride 2 movement and adding 1 pixel of padding on each side (in grey) to maintain the same output shape as the input (in white).

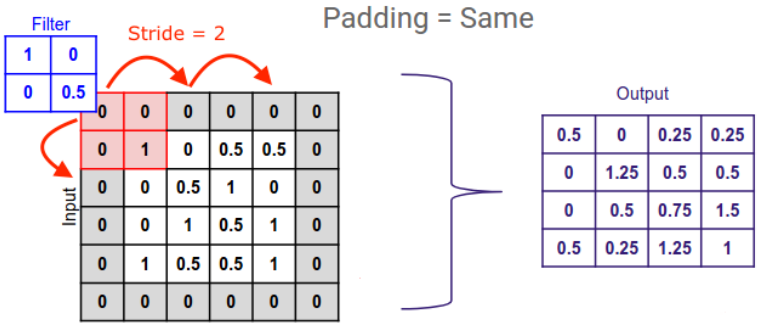


Figure 2: Convolution step with same padding and stride 2 [12].

Each convolutional unit also includes a bias term, that gets added after a convolutional pass is done and (as kernels do) is also tuned by training.

[11] Source: `http://www.davidsbatista.net/assets/images/2018-03-31_dpln_0412_cnn.png`
[12] Source: `https://labs.bawi.io/deep-learning-convolutional-neural-networks-7992985c9c7b`

Activation function

Mimicking the actual biological action of neuron firing, activation functions are placed after a convolutional (or a fully connected) layer in order to perform a non-linear transformation on the data, so that more complex representations can be built. Despite being several types of activation functions (*tanh*, *sigmoid* etc.), nowadays the most used activation function is the Rectified Linear Unit which transforms any negative value into 0's.

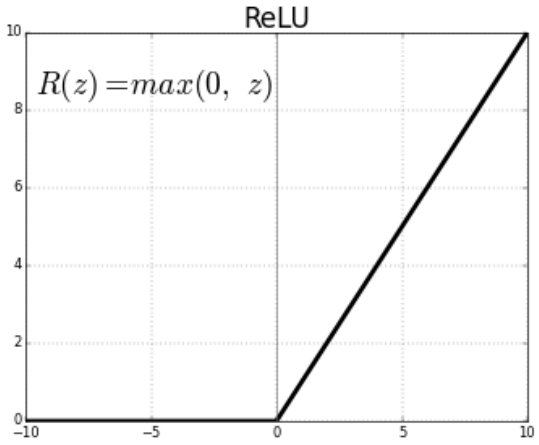


Figure 3: Non-linear ReLU activation

Pooling layer

Usually after a convolutional layer, the dimensionality of the output features might still need to be reduced. However, a naive down-sampling could translate in a loss of some of the patterns w.r.t that the outputs features carry. Thus, being the convolutional units sensitive from where the patterns are located in their input feature maps, down-sampling must be applied with a set of operations that can summarise the patterns found in the up-scaled version of the features. The most common used pooling operations are the average and the max. These operations are applied on each patch (or part) of the features.

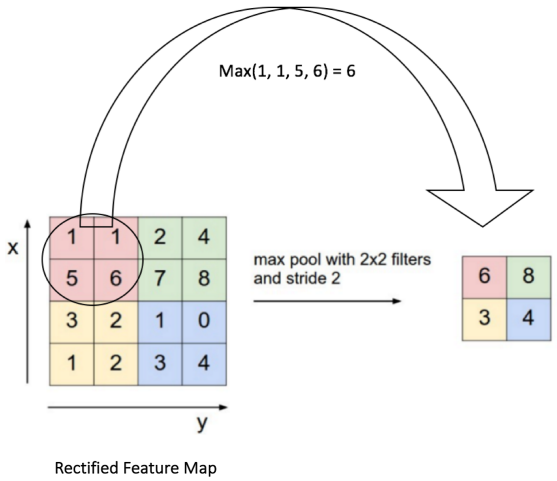


Figure 4: Max pooling of a given output feature [13]

Pooling layers also apply a pooling operation on each of the feature maps / output channels. Similar to convolutional units, pooling operations also move through the feature maps with a given stride (stride 2 in fig. refimg:maxPooling) and map to a certain region sized as the specified pooling size (2×2 in fig. refimg:maxPooling). On an input being 4×4 with a pooling size of 2×2 the output shape would be reduced to 2×2.

Fully connected layer

Relatively similar to the convolutional units, fully connected units follow the same principle as in the Multilayer Perceptron. Given an input, fully connected units perform matrix multiplication operations between the inputs and weight matrices. Using the same strategy as the convolutional units, each fully connected unit includes a bias term that gets added to the activation value (output features).

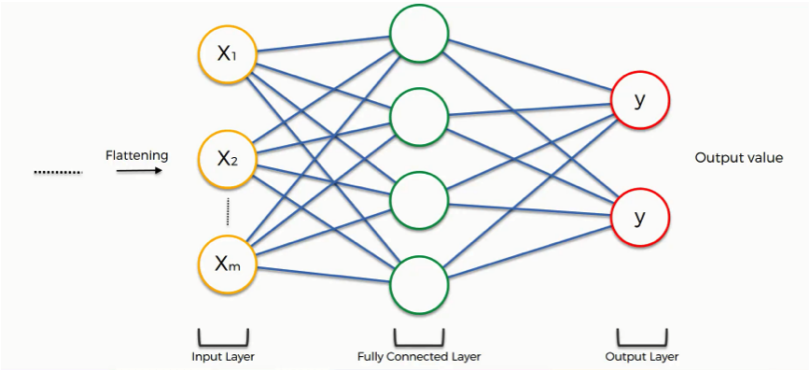


Figure 5: Structure of a fully connected layer

Those weight matrices represent how much importance (or strength) each unit gives to

[13] Source: `https://ujwlkarn.files.wordpress.com/2016/08/screen-shot-2016-08-10-at-3-38-39-am.png?w=988`

an actual input, being reflected at the output. At a fully connected layer, weights are understood as matrix of dimensionality $input\,size \times fully\,connected\,units$. In a traditional MLP and fully connected networks those matrices would represent the vast majority of trainable parameters given its dimensionality.

Softmax / prediction

In order to output a prediction, most of the time a softmax activation function is used to transform the output values of the given layer into an array of probabilities that add up to 1. The index in that array with major probability is chosen as the predicted class of the network.

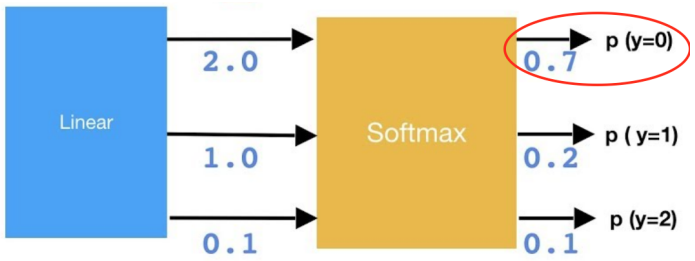


Figure 6: Softmax into 3 decision classes and chosen prediction.

Simple CNN layer organisation

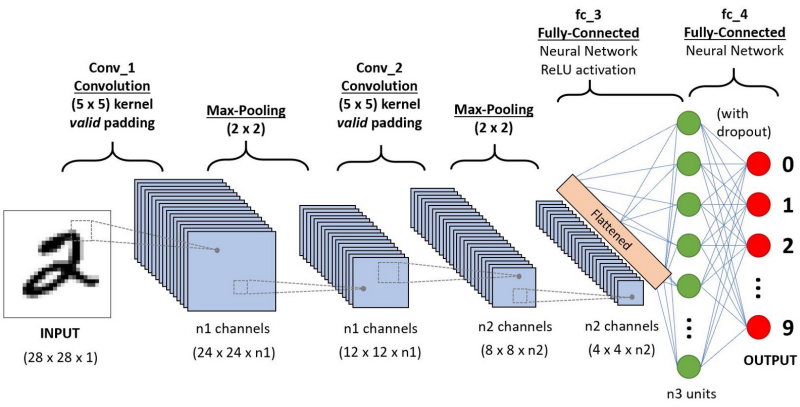


Figure 7: Simple convolutional neural network architecture [14]

Seen in the previous figure 7, being a digit prediction network, gets as input an digit image and outputs a class from (0 to 9). The network structure is based on two convolutional layers (that apply Max pooling after each one of them) on top of two fully connected layers resulting in a output dimensionality into 10 classes.

[14] Source: `https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way`

Training through gradient descent. Training several layers through error propagation

Once predictions are made (whilst training), predictions error (or loss) is computed by comparing the actual predicted with a pre-generated true class (whilst processing the input dataset) to generate a given metric. This metric usually known as categorical cross-entropy (having more than two classes to predict) or binary loss.

The error (or loss) can be written as a mathematical formula (the cost function) that evaluates all the transformations that the input data has suffered (while being forwarded through the network). Therefore reducing the error can be understood as an optimisation problem. By means of automatic differentiation, starting from the cost function (and using the chain rule) the partial derivatives of any given model parameter w.r.t the cost function can be obtained. By doing so, on each training iteration, weights are updated after an entire batch is forwarded from the input (forward pass) using a optimisation algorithm (e.g. SGD) to direct the function to a minima.

1.3 Model architecture

The neural network model Inception V1 [2] (a.k.a GoogleNet) it's the one chosen to be used in all the following experiments (and results) since:

- Its a very successful model, providing particularly visual interesting patterns.
- Has a reduced amount of layers, (compared to other models such ResNet [3])
- Having won the ILSVRC '14), GoogleNet bringed several improvements over previous award-winning models (e.g: AlexNet [4], LeNet [1] etc.) such as the innovative inception layers.

1.3.1 Inception layers

Having a total of 22 layers, GoogleNet was one of the first to feature a multi-size convolutional layers (a.k.a Inception layer), which feature 1x1, 3x3 and 5x5 kernel size convolutional units.

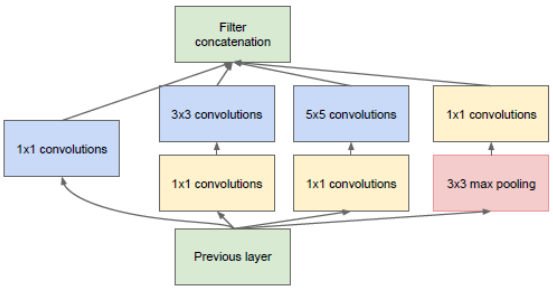


Figure 8: Structure of an Inception Layer [15]

[15] Source: [2]

Whilst maintaining the same output size as the input, Inception layers feature 1×1 convolutions before the actual 3×3 or 5×5 convolutions. By convolving on a higher input dimensions (channels) compared to their outputs, 1×1 convolutional units are able to represent and embed the input channels (#input channels) to a different amount of output filters without modifying the input shape. Thus effectively achieving a channel pooling. A simple example example can be found in figure 9, where 32 input channels can be reduced (or increased) into n output channels.

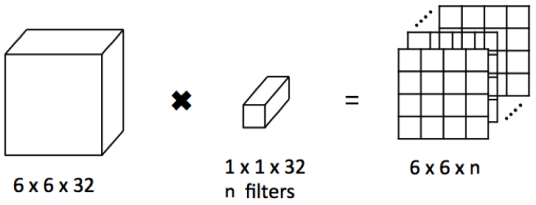


Figure 9: 1x1 convolution units for channel pooling

By using those units as channel pooling units, the number of operations and memory footprint is considerably reduced since the following 3×3 or 5×5 convolutional units would convolve around a much smaller number of channels.

Nonetheless, different kernel sized convolutional units inside the same layer (3×3 and 5×5) allow for different sized patterns to be recognised.

A full detail on the GoogleNet architecture can be found in the Appendix section 7.1.

2 State of the Art

Over the last years, several techniques have been introduced to help understand how Neural Network (NN) (specially Convolutional Neural Network) *learn*.

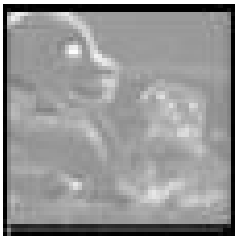
In general, one of the most desired features of a model is its ability to generalise. That is, provide good predictions against unseen data. Without any further measures, the quality CNN could only be assessed during training time by observing the accuracy using the testing and validation sets, or once the model has been trained, testing the model generalisation by using unseen data.

However, with these approaches, no insights are obtained on how the networks reacts to different inputs, obtaining what is commonly known as a *black box*.

Visualising layer activations [5]

Apart from the aforementioned measures to validate a quality training on a neural network, we can get a glimpse of how the CNN are behaving by getting outputs at any layer during a forward propagation given a certain input. Such input should clearly be allowed to be classified.

(a) Input image for the network GoogleNet



(b) Activation at the first layer conv2d0 unit 138

Input visualisation through activation maximisation [5, 6, 7]

Once having pre-trained a model, starting with a random input or a pre-calculated one [8]we can obtain a more precise look on each individual unit by applying gradient ascent, during a certain amount of iterations, on its activations f. This is in order to obtain the input that maximises the most of the output of the neuron.

$$i^* = \arg\max_i f(i, layer, unit) \tag{1}$$

Only a local minima will be achieved for the problem to optimise is usually non-convex.

Since activation maximisation is the simplest method, several types of regularisation can be applied to reduce high frequency artefacts and obtain more realistic examples [9, 10].

[16] Source: [7]

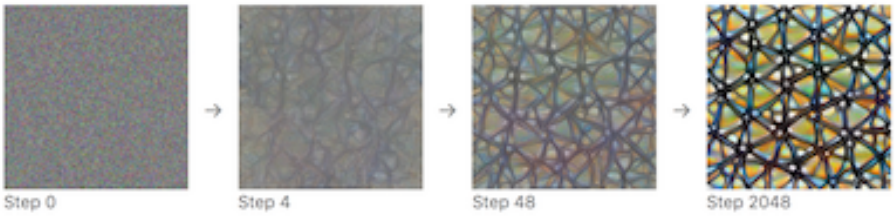


Figure 11: Input timestamps while optimising the activations for the 11th unit in layer mixed4a [16]

Channel and input space attribution to output class [11, 12]

Despite being able observe what the network is learning by visualisation, no insights are provided to answer the network learning decisions based on its architecture, inputs and output classes.

Saliency maps [11] can be computed in order to show which pixels of the input space that contribute the most to the actual output class.

Figure 12: Saliency maps of the upper pictures [17]

However, saliency maps do not take into account which units (channels or even layers) contribute the most to an actual output class decision.

User Interface for CNN interpretability [13, 12]

Combining some of the previous methods, there are already animated user interfaces that provide a graphical interpretability for a given CNN. This not only allows the developer to avoid spending time working on a framework for its CNN architecture but also to get a faster and more complete insight on how the network units affect the output class.

[17] Source:[11]

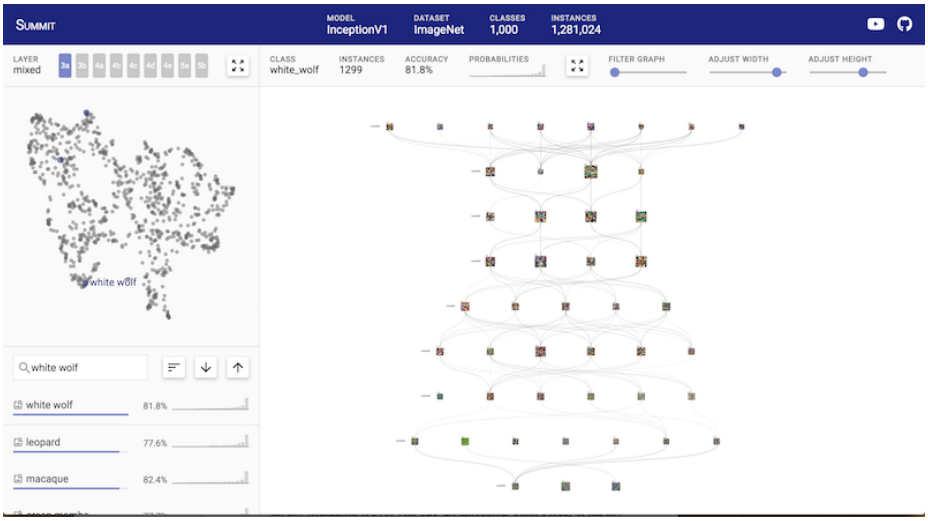
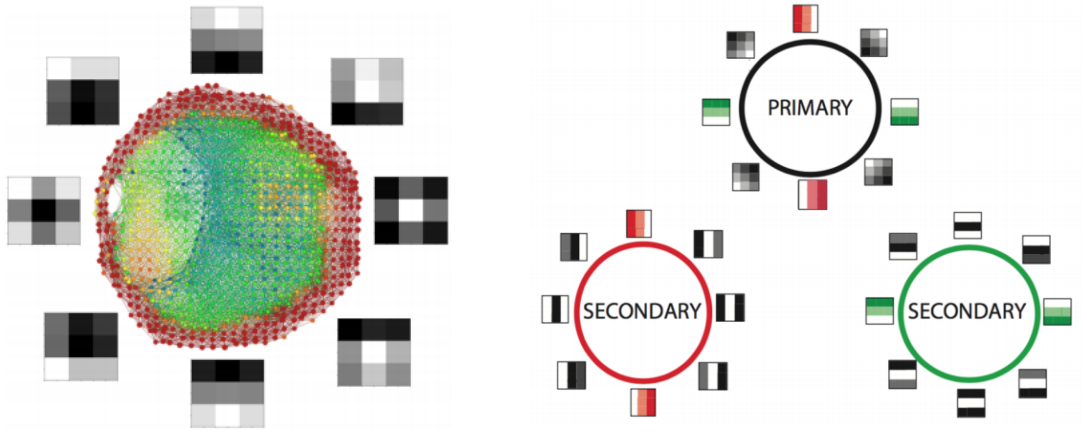


Figure 13: GUI that summit provides [18]

Topological Data Analysis on simple CNN weights Having gained popularity from the last 5 years, Topological Data Analysis (TDA) is starting to be used in commercial application and small academia. Providing a compression of a large set of states into a much smaller and more comprehensible graph representations, topological data analysis, can be used to understand the topological insights on a given data. As a proof of concept of their proprietary technology based on a mapper algorithm, Ayasdi[19] representatives applied TDA on a CNN architecture trained on a MNIST dataset [14] for digit prediction. The resulting shape of the graph (in the left image of figure 14) did show an actual ring. By showing the kernels of the given units found in each of the actual graph nodes, they did see that such kernel patterns (3×3 sized binary images) were actually rotating in each given position of the ring.

[18] Source:[13]

[19] `https://www.ayasdi.com/`



(a) Resulting TDA graph on a given layer of the previous-mentioned CNN architecture.

(b) Image based on the three circles with corresponding transformation of a 3×3 image patches.

Figure 14: Resulting TDA [20]graph on a given layer of the previous-mentioned CNN architecture.

The actual topological representation of the layer, found in the left image of figure 14, was actually providing the entire range of rotations of 3×3 patterns combined in the three circles, found in the right image of figure 14. Having shown at [15] that the primary visual cortex, for humans and mammals, is a detector for edges and lines, they demonstrated that the layer of that given model was actually really mimicking the primary visual cortex.

[20]Source:[14]

3 Methodology

In this chapter a more detailed study on the model neurons will be held. At first, where the methods for the data sources creation and extraction will be explained. Two parts will follow the data sources acquisition, where both supervised and unsupervised methods for learning different neuron behaviours will be detailed.

In order to accomplish the above-mentioned parts, a Python environment will be used. It will consist of:

- Tensorflow as the deep learning framework.
- Lucid as the interpretability and feature visualisation package, on top of tensorflow.
- Scikit-learn as a machine learning package.
- Numpy as a mathematical package for Python.
- Matplotlib and seaborn as visualisation packages.
- keplermapper as a TDA package.

After a brief introduction on the model architecture, several things are needed before conducting any further experiment.

3.1 Getting neuron weights & biases

The package Lucid [16] provides several pre-trained models. Said package which relies on tensorflow[17] as the deep learning framework to provide a better neural network model interpretability. Being fairly accessible to the end user, weights of any given model can be extracted by interpreting the protobuf [21] file included in them, as seen in algorithm 1.

Algorithm 1: Extracting neuron weights

```
Pre-trained model protobuf file protbuf_file List of neuron layer weights
with new Tensorflow Session:
weights = []
with open(protbuf_file) as graph:
tensorflow.import(graph)
weights_nodes = [node in graph_nodes that node.operation == 'Const']
for layer in weights_nodes do
    convolutional_units = ['1x1_w', '3x3_w', '5x5_w', 'pool_reduce_w']
    if layer is an inception layer and is not a bias layer and layer contains
     convolutional_units then
        weights.append(layer)
return weights
```

Given a *tensorflow* session, the *protbuf* file can be interpreted as a graph representing our model. Nodes that are not operations, such as activation functions, are called *Constant*

[21] Being a serialisation format for structured data from Google.

and store the weights of each layer as *tensorflow* tensors. The final weights can be extracted by accessing the tensors.

3.2 Optimising input images for maximum activation values

Starting with some random noise, and by process of optimisation on the activations, we can obtain an image that maximises the activations on a given part of the CNN model.

Depending on the objective to accomplish, different parts of the model or optimisation objectives can be used. From optimising a single neuron in a single channel, to an entire layer or even an entire output class. This allows us to really understand what the model, or part of it, is looking for at very different levels.

As this thesis title of this specifies, optimisation will be applied at the neuron level. In order to get the closest approximation to its behaviour through optimisation, it has been decided to not only search for what excites a neuron the most (maximisation), but also what generates the minimum activation value. In order to do so, optimisation will be applied without taking into account the activation function, as can be seen in algorithm 2. This is always applied after a convolution during forwarded pass, since its characteristics of transforming negative values into 0's would interfere with the previous mentioned purpose.

Algorithm 2: Optimised Images through neuron activations

Input: Unit to optimise
Output: Image corresponding to that unit

```
inputImage = randomNoise
iterations = 0
threshold = 2048
while iterations ≤ threshold do
    activations = forwardStep(inputImage)
    inputImage = adamOptimizerStep(δactivations / δinputImage)
return inputImage
```

As precautionary measures to obtain images that can truly relate what a neuron is looking for, regularisation must be applied during optimisation in order to avoid high frequency artefacts in the resulting image or adversarial examples[18]. Although a strong regularisation would translate into ideally realistic examples, it could end up being misled into a wrong representation of what really a neuron is activated. Therefore, all images have been computed with several optimisation measures, which can be found in 7, and that lucid provides.

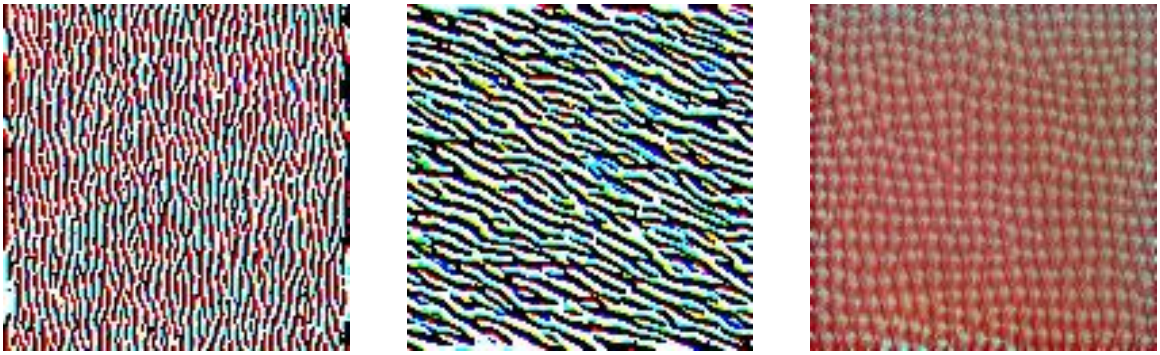
4 Analysing neuron weights and optimised input images

At first, a supervised approach will be put in practice to find different neuron behaviours that can be humanly describable. This involves reducing the complexity on the possible patterns found in the neurons to a certain amount pattern shapes such as circles, diagonals, lines, borders and colour patterns, among others.

A second approach will follow, where unsupervised methods will be used to find groups of neurons that follow a similar behaviour. Where possible, the characteristics of each group will be analysed.

Data selection

As shown in the introduction, GoogleNet architecture is compromised of 22 layers being only 9 of them based on the inception layer. Since we are only interested in the convolutional layers it has been decided to use the convolutional units inside the inception layers as the data for any further experiment. Despite having three convolutional layers before the actual inception layers, shown in the full GoogleNet architecture in fig. 39, it is seen in the optimised images of those first convolutional layers that they do not provide as meaningful patterns as the inception layer that follow them.



(a) Edges patterns on one of the first three convolutional layers unit.

(b) Complex pattern found the the inception layer units

Coupled with this, Inception layers provide a much interesting structure of convolutional units that use different kernel sizes. Therefore, it has been decided to focus on the convolutional units found in the nine Inception layers, ranging from *mixed3a* to *mixed5b*. Once the

layers are decided, it is worth mentioning the characteristics of the 1×1 convolutional units (found in the appendix 7.1) despite providing interesting characteristics, due to its given inner structure (having only 1 element as their kernel) could be understood that they lack the ability to retain patterns that 3×3 and 5×5 units have.

4.1 Supervised approach

After having generated all the optimised input images for the entire network and having decided to select and work solely on the Inception layers, a given set of steps are going to be used in order to obtain, decide and evaluate each of the possible patterns that the neurons have.

An initial visual exploration on the optimised images will be done. This will focus not only on the maximum optimised images but also on the optimised images that minimise the neuron activations. Parallel to that, Distill provides a wide appendix of pictures [22] , which are present in the dataset used to train this NN that excites or activates the most and the least of each unit present in the *layers*. This will help diagnosing the patterns that are not entirely visible in the optimised input image alone.

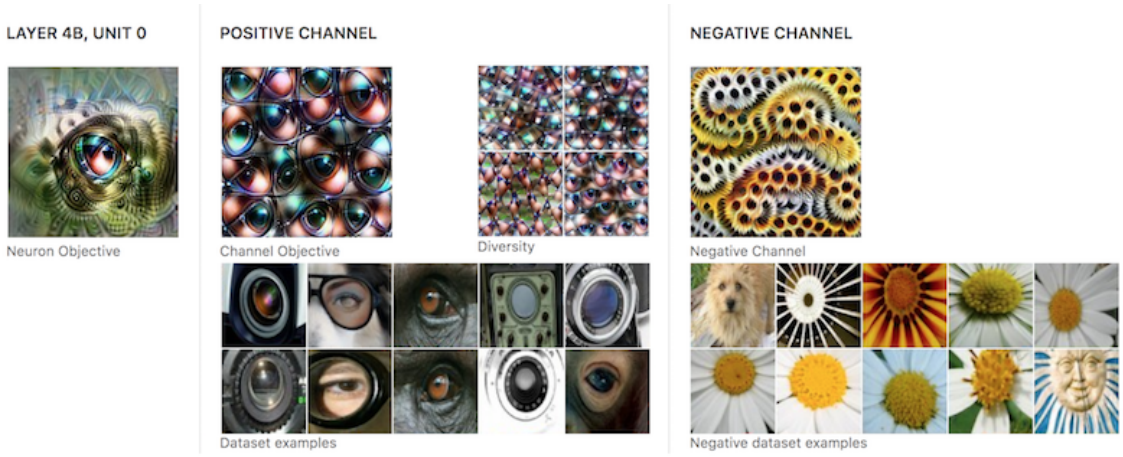
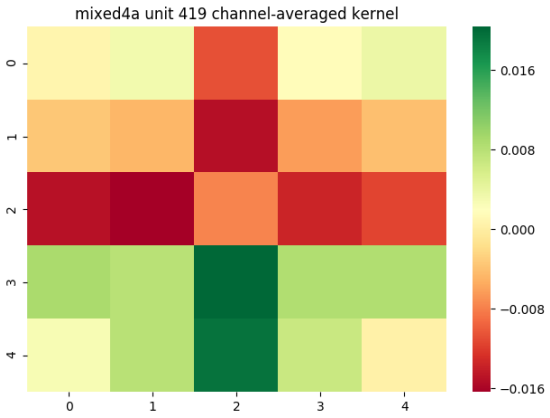


Figure 16: Distill appendix on mixed layer units

After a visual inspection on the optimised images and the most representative dataset examples for each unit, each unit weights will be analysed following a double measure. First, the high amount of input channels on a given convolutional unit weights would suggest to compute a histogram of each of the neurons channels or a heat map of the channel-wise averaged kernel values. Insights on how the patterns are placed could be found by taking a looking on how the positive values against the negative are situated inside the kernel. Comparison of the pattern against the previous mention set of images is gonna be useful to identify which is the pattern.

[22] Available at `https://distill.pub/2017/feature-visualization/appendix/`



(a) Channel averaged kernel matrix for unit 419 in layer *mixed4a*

(b) Images that excite the most unit 419 in layer *mixed4a*

Figure 17: Circle patterns matching the positive values of the (averaged) kernel values for unit 419 in layer *mixed4a*

Finally, another measure to possibly consider would be the variance that is present in the units weights kernels. Under the assumption that a same exact pattern pattern might be repeated in several images, the variance of the pattern, represented into the kernel, could be lower than the rest of the parts of the kernel matrix, by taking into account that the background might change.

4.1.1 Colour blind neurons

By visualising the generated optimised input image, we can observe that some neurons share similarities regarding the colour scheme. Despite having completely different patterns, those neurons seem to be memorising the same pattern without taking into consideration the colour, mainly because of the inputs while training.

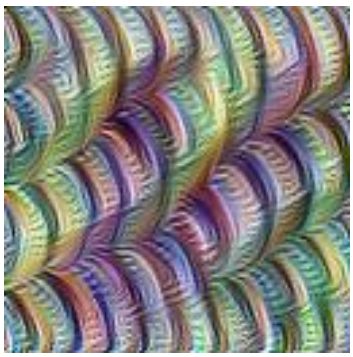
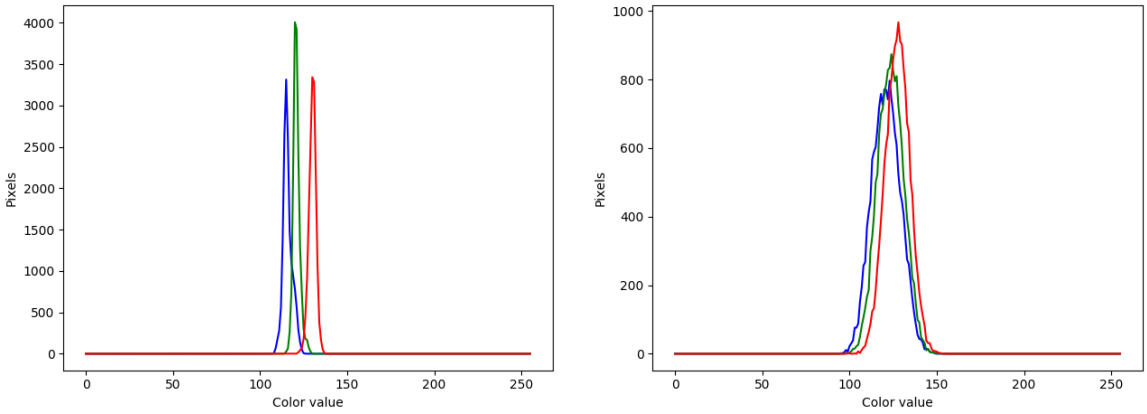


Figure 18: Colour blind unit 104, layer mixed3a.

By plotting the histogram of the averaged RGB components of the colour blind considered images vs the rest, we have found that those images that have relatively the same density peak on all of the components could be classified as colour blind.



(a) Non colour blind labelled RGB histogram (b) RGB histogram of the colour blind images

Figure 19: RGB density comparison between non colour blind and color blind images.

4.1.1.1 Distribution across layers

In addition to its behaviour, the amount and percentage of colour blind neurons on each layer is computed in order to understand if there is any relation.

Layer	**mixed 3a**	**mixed 3b**	**mixed 4a**	**mixed 4b**	**mixed 4c**	**mixed 4d**	**mixed 4e**	**mixed 5a**	**mixed 5b**
#	40	2	1	0	0	0	0	0	0
$(\frac{\#}{layer\ size})$ %	15.62	0.41	0.19	0	0	0	0	0	0

Table 1: Distribution of colour blind neurons across layers

Having performed a supervised approach, we can objectively say that the first layer contains most of the colour blind units. However, a colour blind classifier might give different results due to higher abstraction that images in deeper layers provide, which can be observed from fig. 40 onwards. That level of abstraction might complicate any further discovery by means of visualisation of colour blind units.

4.1.2 Border aware neurons

Apart from having found colour blind units, another behaviour can be found as patterns that are seen in any of the image borders. Those patterns appear to be locate2d on the image borders and are different from the rest of the image.

Figure 20: Optimised input image for border aware unit 407, layer mixed4b

4.1.2.1 Distribution across layers

After having applied a supervised approach by visually analysing the optimised input images, their distribution through his layers has been computed.

Layer	**mixed 3a**	**mixed 3b**	**mixed 4a**	**mixed 4b**	**mixed 4c**	**mixed 4d**	**mixed 4e**	**mixed 5a**	**mixed 5b**
#	5	8	34	17	10	11	2	0	0
$(\frac{\#}{layer\ size})$ %	1.95	1.66	6.69	3.32	1.95	2.08	0.24	0	0

Table 2: Distribution of border aware neurons across layers

Whilst having found lower amount on the deeper layers (*mixed4e* to *mixed5b*, due to the higher level of abstraction) most of the border aware unit have been seen in the *mixed4* layers, which seem to provide interesting amount of visual meaning.

4.1.2.2 Channel-wise weights comparison

Once gathered, each sample of border aware unit is analysed using its weights, having a structure or dimensionality such as the one seen in figure 21.

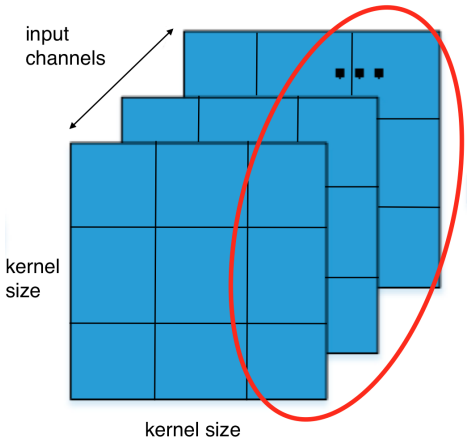
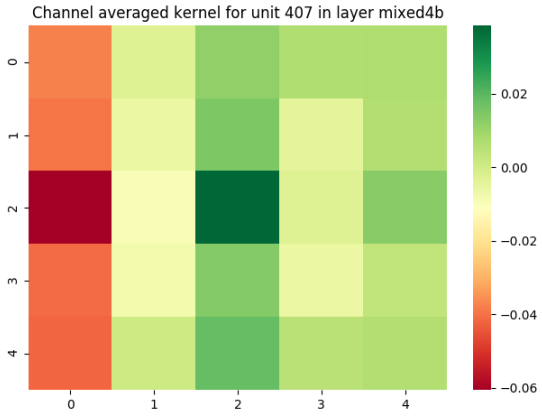


Figure 21: Graph representation of a neuron's weights

As explained at the beginning of this section, channel averaged weights will be used to understand how the pattern might be arranged in the image. Since wanting to find patterns that reside at the borders of the image, comparison between values of the kernel (being column-wise or row-wise) will be established by how the pattern is visually represented in the input image [23].

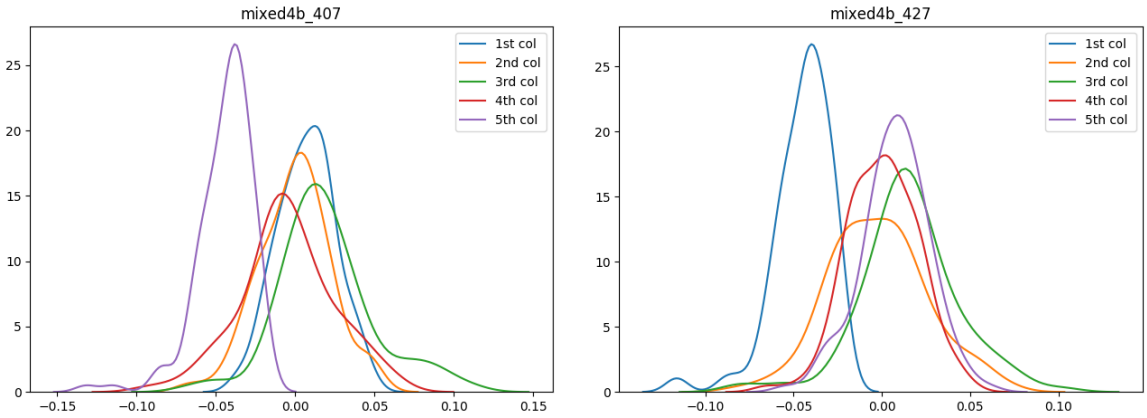
(a) Optimised input image *mixed4b* 407).



(b) Averaged kernel values for border aware unit 407 in layer *mixed4b*).

Having applied the aforementioned comparison, it has been seen that border aware unit have a quite different density peak on the columns and rows that belong in one of the extremes of the kernel matrix, on each of the channels, compared to the rest of them.

[23] A vertical (or column-wise) comparison of given the unit in figure 20, since we find a vertical pattern on the left side of the image.



(a) Left side border aware neuron (*mixed4b* 407). (b) Right side border aware neuron (*mixed4b* 427).

This phenomena not only allows us to create an unsupervised classifier for the type of neuron, but also encourages us to experiment with the dual behaviour that this type of neurons provide.

4.1.2.3 Analysing border neuron duality

Given a sample input image, we try to find how the border aware units behave, assuming to have two differentiated behaviour on its weights. In order to do so, the weights corresponding to figure 20 are used as a border aware sample in a forward pass with a replicated structure of the chosen part of the layer corresponding to that unit. That replicated structure can be seen in figure 24.

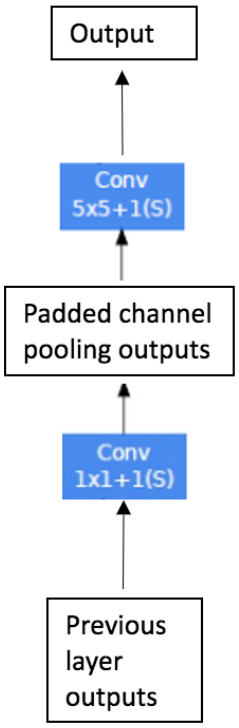
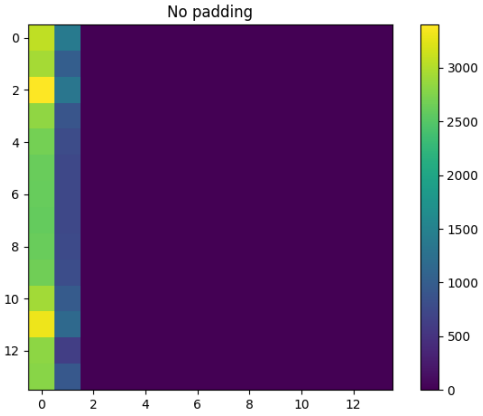


Figure 24: Replicated structure for border neuron duality analysis.

This replicated structure mimics the actual layer *mixed4b* of the previous neuron (unit 407) allowing the inputs to be changed, instead of using the outputs of the previous layer in question mixed4a. Situated after the channel pooling units, unit 407 is part of the 5x5 convolutional units, inside Inception layer *mixed4b*, represented in figure 8. Therefore, in order to simulate a plausible forward pass, inputs must be replicated through the channel pooling units prior to the actual border aware unit.

The forward pass uses as input images that are identically sized (14 by 14 pixels, as the actual inputs on the given layer *mixed4b*) that featured outputs from the previous layer (*mixed4a*) and guarantee that the resulting output from the actual unit (layer *mixed4b* unit 407) would be a pattern in the border, similar from how the pattern is represented in the optimised input. For example, vertical in case of a vertical pattern image, such as figure 20.

(a) Input image forwarded to the network



(b) Activations (layer *mixed4b* unit 407) for the previous input image

As stated in the Appendix, convolutional units use same padding configuration, which adds empty (0's) pixels in each border of the image in order to guarantee that the input dimensions (14 by 14 pixels) are maintained in the outputs. Therefore, we aim to change the neuron behaviour by modifying the padding value. Being initially filled with 0's, the actual padding is changed with random noise extracted from the range of values that the previous layer *mixed4a* has on its outputs. Therefore, not *upsetting* or distorting too much the output values.

By doing so, we aim to trick the neuron into thinking that the actual border is still part of the actual image, resulting in an empty activation image. In other words, meaning that the neuron does not fire anymore since no border is found.

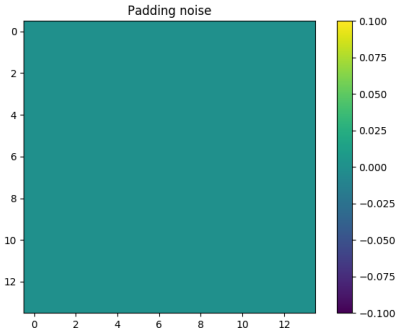


Figure 26: Activations of border aware unit 407 in layer *mixed4b* after changing the padding.

4.1.3 Validating unit behaviour from weights

Having found border aware neurons that replicate the pattern in their kernels, the focus will be set on finding other units that also show a clear correlation between their behaviour and their weight values.

A first example can be given by looking at neurons that provide a diagonal shape in their optimised input images (e.g. in figure 27).

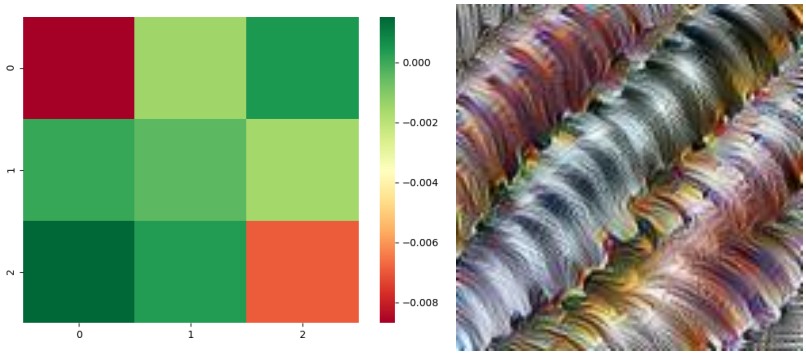


Figure 27: Averaged kernel and optimised input image for neuron 235 in layer *mixed4a*

By combining kernels of the given unit into a single one, by averaging them, we can obtain a kernel that clearly provides a separation between clearly negative values positive ones that shares the same shape direction as the optimised input image. In order to validate it, images that provide the highest activations show the same diagonal shaped pattern, available in figure 28.

Figure 28: Images with higher activation values for layer *mixed4a* unit 235

Slightly more complex patterns seem to also represent the same idea. The following optimised input image (in fig. 29) show two different directional patterns.

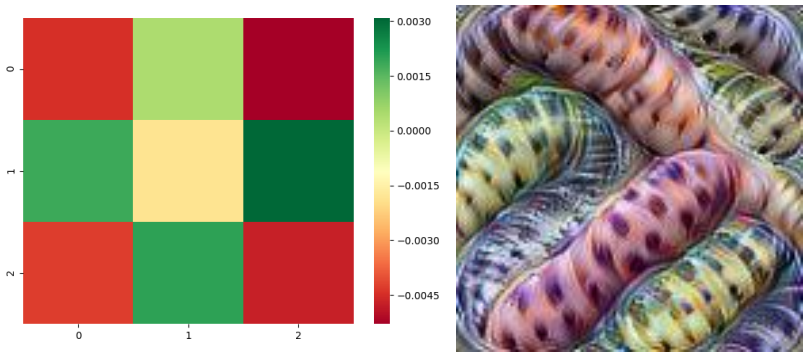


Figure 29: Averaged kernel and optimised input image for neuron 304 in layer *mixed4a*

However, the averaged kernel matrix also shows two diagonals in their positive values. Images that activate or excite this unit as shown in fig. 30, also confirm the same theory by clearly displaying two different diagonal patterns.

Figure 30: Images with higher activation values for layer *mixed4a* unit 304

Taking into account negative kernel value Despite only having having taken patterns into account by observing the positive values in their kernels, the negatives ones also play a role at characterising the neuron.

As a clear example, figure 31 shows a T-shape starting from both sides and ending at the top of the image.

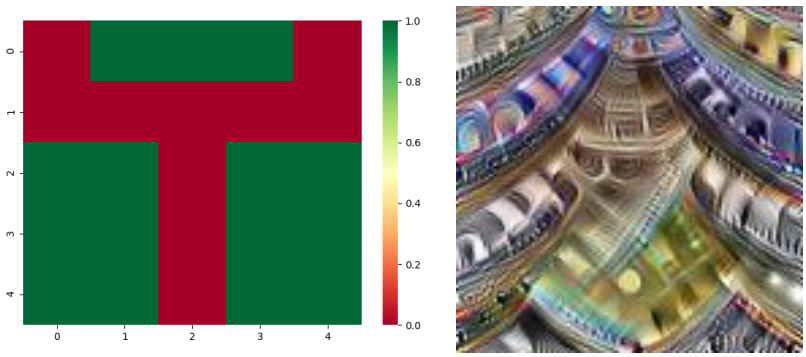


Figure 31: Averaged binarised kernel and optimised input image for neuron 428 in layer *mixed4a*

Observing the binarised average kernel for that unit, we can distinguish a clear pattern for the positive values, seen in green, that start from the sides of the image to the actual top of it, separated by negative values, which are shown in red.

Not only images that excite that neuron, as seen in fig. 32, have patterns shaped similarly to the positive values, but also images that activate the less the same unit share the same pattern as the negative values.

Figure 32: Images with higher activation values for layer *mixed4a* unit 428

Using a supervised approach, the study of the neuron weights has been really useful to provide insights of them by having discovered the previous behaviours. We have seen that units map the pattern, in the images that they activate to, to their kernels and finding different associations between negative and positive values of the unit kernels. However, there is no *big picture* in the sense that there does not exist a generalisation on how units are distributed through the layer of the architecture nor if there are any groups.

4.2 Unsupervised approach

Having found that units map the pattern that they activate to their weights and the lack of a general idea on how weights are distributed, we aim to perform an unsupervised approach that will consist of the two next steps. From one side, classifiers for the previous types of neurons, found by having followed a supervised approach, will be built and discussed. From the other side, new behaviours will be searched for by means of several approaches, such as dimensionality reduction, clustering and TDA, using both the neurons weights as main input. If required, data visualisation will be used to provide a better explanation as well as insights.

4.2.1 Colour blind images classifier

After having found a way to classify images that could end up representing colour blind unit, a simple classifier is built. The steps followed can be seen in algorithm 3. Using opencv we separate each optimised input image into its three main channels (Red, Blue, Green) and compute the density peaks. By measuring the standard deviation on them, images that have near identical density peak values (having its Std below a certain *threshold*) are classified as colourblind.

Algorithm 3: Colourblind images classifier

Input: Optimised images
Output: List of images that are considered colour blind
`/* pixel colour std that is considered */`
threshold = 3
colourBlindImages = []
for *image in Optimised images* **do**
    if *std(channelDensityPeaks(image))* $\leq$ *threshold* **then**
        colourBlindImages += image
return colourBlindImages

Having found a relative amount of units, which can be seen in table 3, a sensitive value for the *threshold* has been provided, since inadequate values could lead to biased results. A small value of 2 has been used as standard deviation. Further consideration must be taken into account, since no relation was found between unit weights and their respective input optimised images. Therefore, it is assumed that the variety of colours in the samples that makes those neuron activate is responsible for the actual colour-blind behaviour.

Distribution across layers

Layer	mixed 3a	mixed 3b	mixed 4a	mixed 4b	mixed 4c	mixed 4d	mixed 4e	mixed 5a	mixed 5b
#	39	41	38	33	14	31	42	36	62
% ($\frac{\#}{layer\ size}$)	15.23	8.54	7.48	6.44	2.73	5.87	5.04	4.32	6.05

Table 3: Distribution of colour blind neurons across layers

Despite being slightly different from the supervised approach table, found in table 1, layer *mixed3a* still holds most of the colour-blind units, followed by the next layer *mixed3b* and *mixed4a*.

4.2.2 Border aware images classifier

Despite having found a correlation between patterns that can be found in images that activate certain units and their weights, attention it is set into the special case where patterns are found in the image borders. As described in the supervised approach for the same type of units, a given neuron can be considered border aware if its haves different weights in a given border column or row (corresponding to a vertical pattern or an horizontal on) as the algorithm 4 describes. In order to provide a measure of distance between rows (or columns) values in weights for that given range are compared using a two sample Kolmogorov-Smirnov test. This test provides a metric of dissimilarity based on the supreme difference of the cumulative probability between them.

Algorithm 4: Border aware neuron classifier

Input: List of neuron weights
Output: List of labelled neuron weights classified as border-aware

```
1  border_aware_neurons = []
2  for Neuron in layers do
3      for Pair of columns in Neuron weights do
4          columnDissimilarity[Pair] = computeKSdistance(Pair)
5      for Pair of rows in Neuron weights do
6          rowDissimilarity[Pair] = computeKSdistance(Pair)
7      maxColumnDissimilarity = max(columnDissimilarity) - min(columnDissimilarity)
8      maxRowDissimilarity = max(rowDissimilarity) - min(rowDissimilarity)
9      if maxRowDissimilarity ≥ threshold & maxColumnDissimilarity ≥ threshold
        then
10         borderAwareNeurons.append(Neuron)
11 return borderAwareNeurons
```

Since a given parameter *threshold* is introduced in order to account for how much different one row (or columns) must be different from each other, special caution must be taken in order to tune this parameter. Whilst lower *threshold* might not give too much difference

between rows (or columns), high values could end up forbidding the classifier to find any border aware unit. A relatively high value of *threshold* has been used (0.5) in order to make sure that all of the found units can be validated as true border aware units.

Distribution across layers

A total of 25 units that fall below the previous threshold used 0.5 have been found, distributed in the layers as the table 4 specifies:

Layer	**mixed 3a**	**mixed 3b**	**mixed 4a**	**mixed 4b**	**mixed 4c**	**mixed 4d**	**mixed 4e**	**mixed 5a**	**mixed 5b**
#	2	3	2	5	2	2	7	1	0

Table 4: Distribution of border aware units across layers

From the previous 25 neurons, 5 of them are worth mentioning (['mixed3a_67', 'mixed3a_190', 'mixed3b_390', 'mixed3b_399') since [13] also found them as non semantic and border aware [24], thus validating the previous approach used.

4.2.3 Kernel Clustering

Given the difference in kernel sizes on the Inception layers (having 1×1, 3×3 and 5×5 convolutional units), it has been decided to compute different clustering instances for each one of them (only taking into consideration the two former ones due to the characteristics of 1×1 units explained in the Appendix, section 7.1). This decision has been taken after observing that, apart from having different kernel sizes, convolutional units have different input channels depending on the layer (shown in fig. 38) thus not allowing to consider the entire layers as input for a clustering step. Following this strategy, the dimensions of the data for a clustering step could only be dependant of the kernel size and the number of input channels on each layer.

4.2.3.1 Unit wise approach

Having seen that units keep their resemblance in their weights from the actual images that activate them, it is decided to use the weights of each unit to perform clustering in order try to obtain the different behaviours that each layer consists of.

Data set creation for clustering

Being specified to use each units weights as sample, we follow the next steps in order to create a pair-wise distances between the units weights of a given layer (also shown as part of the actual algorithm 5):

- Each unit weights are normalised to mean 0 and variance 1.

[24] Section 8.4 Finding Non-semantic Channels

- Each unit weights having dimensions $kernelsize \times kernelsize \times inputchannels$, is reshaped into a 2-Dimensional matrix with dimensions $kernelsize^2 \times inputchannels$ to simplify its dimensionality.
- After that, the dissimilarity (based on an euclidean distance) of each pair two different units weight (in a given layer) is computed using the euclidean norm of its subtraction.

Once having computed this matrices for each layer, those will be used as input data for further methods.

Clustering algorithm

UMAP[19] is applied prior any clustering in order to reduce the dimensionality that the units might have (taking into account a unit structure of $\#units \times kernel\_height \times kernel\_width$) into a 2-Dimensional space, not only suitable for visualisation but also for decreasing the computational time of any further clustering.

UMAP consists on a two phase algorithm that constructs a representation in lower a space of a topological representation of the input data. The first phase consists of constructing a fuzzy topological representation, essentially as described above. The second phase is based of the optimisation of the low dimensional representation to have as close a fuzzy topological representation as possible as measured a by cross entropy measure.

Clustering will be done in two steps. At first hierarchical clustering will be used to get insights on how many clusters the data can be partitioned in (by considering the height between different cuts in the dendrogram). Since having a low amount of data to cluster (at most 500 elements in the worst case) it is decided to use Spectral clustering against other clustering algorithms (e.g K-means) since it can deal with more complex data. Therefore, Spectral clustering will be used with the decided number of clusters shown in the computed dendrogram.

All this steps can be found in the algorithm number 5.

Algorithm 5: Neuron weights clustering

Input: layerWeights, kernelSize, numberOfClusters, (optional), dimensions
Output: List of clusters assigned to each neuron

```
distanceMatrix = []
for pairOfNeurons in layerWeights do
    if pairOfNeurons have kernelSize then
        normalizedNeurons = meanCenterScale(pairOfNeurons )
        neuronsDistance = euclideanMatrixNorm(subtract(pairOfNeurons))
        distanceMatrix[pairOfNeurons].append(neuronsDistance)
projectedUnits = Umap(distanceMatrix, dimensions)
dendrogram, numberOfClusters = HierarchicalClustering(distanceMatrix, 'ward')
clusters = SpectralClustering(distanceMatrix, numberOfClusters)
return clusters;
```

Validation of clusters

Once clusters are obtained, a method in order for cluster validation is going to be used when necessary.

1. For each of the given units in a cluster, their averaged kernel values are going to be computed. Those kernels will be saved and displayed as a heat map in order to be visually understandable. Positive and negative values of the actual kernel will be interpreted based on a concrete colour map to represent them, found in figure 33. Positive values will fall under red to white colours, white negative ones will fall under lila to black.
2. After that, an unsupervised approach will be used in order to determine, at each position of the averaged kernel matrix, the most repeated sign (negative or positive) in that position.
3. The units that meet the previous criteria are marked as a possible candidates to represent those clusters.
4. Finally, a visual inspection is going to used in order to validate the previous steps.

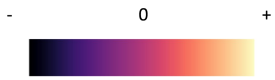


Figure 33: Range of colour for the averaged kernel units

After that, at each given layer the kernels that represent the most each cluster will be displayed in order to aid with the cluster description. An example for the following layer *mixed5b* can be seen in figure 35.

In order to be concise, only the analysis on certain given layers that is relevant for the conclusions in this section is shown. The results for the rest of the layers can be found in the appendix, section 7.2.

3×3 kernel units in layer 5b have been separated into four clusters, shown at the left image in figure 34.

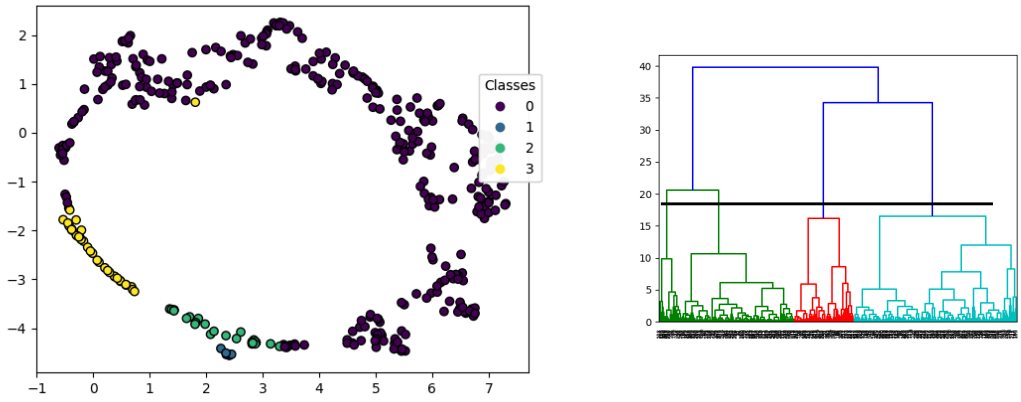


Figure 34: 2D embedding & Hierarchical clustering dendrogram of the average kernels with kernel size 3×3 in layer *mixed5b*

Cluster num.	0	1	2	3
# units	317	6	27	34

Table 5: Size of each cluster in layer *mixed5b* for units with kernel size 3×3

Having a big cluster (as shown in table 16) we can extrapolate that cluster as the actual layer behaviour, which has respectively the following characteristics and representation (observable in 35):

- Units corresponding to the first and second cluster do have a negative value in the centre position on the kernel and have up to two rows of negative values.

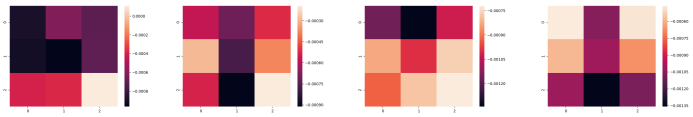


Figure 35: *mixed5b* 3×3 kernels, each representing a cluster

Considerations after clustering

After having applied a dimensional reduction for visualisation and a clustering approach, there are some considerations that must be taken into account:

Shape found in 2D embeddings Once having computed a lower dimensional representation of each given layer, cluster divisions into their embedded 2D units haven displayed as scatter plots in order to help the actual explanation. Nonetheless so far the shape of the actual embedding wasn't considered (or discussed) and could be playing a greater role

contributing to the actual behaviour of the architecture.

From one side we have found that the majority of layers do not result in any known shape at the time their embeddings are computed, which actually end in a noisy and / or shapeless scatter plot (seen in the figures 55 62 and 73). This could be understood as the actual data being spread on the actual feature space, thus indicating that points might be part of one and only big cluster.

On the other side, on the latest 3×3 and 5×5 kernel layer 2D embeddings (since the number of convolutional units reaches its maximum in the deepest layers of the architecture), we can observe that higher layers start to show an actual circular pattern (seen in the figures 67, 73 and 80). Therefore, an actual clustering approach could be unnecessary due to it.

However, these two considerations allows us to infer that the layer weights in the GoogleNet architecture might have a concrete topology in their inner structure.

4.2.3.2 Channel wise approach

Seeing the results found by the unit-wise clustering approach, we decide to inspect the weights in a more fine-graded level. In order to do so, we contemplate each of the channels inside a unit weights as a separate sample. The objectives at this point are two-fold:

- Be able to learn the characteristics of the kernels that are learned. Embedded space coordinates will be shown by layers to check for any correlation between types of channels and layers.
- Have a general representation of the space that the layers (as sets of kernels) provide and (if necessary) apply clustering methods to provide groups of channels.

Topological Data Analysis (TDA)

Due to the high dimensionality of the weights in a given layer and the amount of samples (contemplated since a sample is considered as a single kernel in a unit), it has been decided to use TDA to explore its structure. In python, an outstanding package KeplerMapper [20] (that contains implementation of the Mapper algorithm [21]) is going to be used.

From a high level point of view, the mapper algorithm is often used to create a graph from data that allows to get insights from the data on a topological view. In order to do so, the Mapper algorithm follows this steps:

Data projection through dimensionality reduction In the data projection step (as the title mentions), the whole dataset is projected into a lower dimensional representation. For visualisation measures, the output dimensionality is usually constrained to at most 3 dimensions.

Covering the projection Once the projection have been computed, the resulting projected space is separated into a set of bins (or intervals) on each dimension with a certain overlapping between each consecutive one (e. g: A 2 dimensional projected space with 10 bins is going to be split into 100 quadrants, 10 bins for each dim.).

Clustering by intervals A clustering is going to be applied on each of samples in the previous quadrants. At this point, but original samples in data or dimensionally reduced samples in the projected space can be used.

Graph creation The resulting clusters will be converted into nodes at the resulting graph. Node connectivity will be dealt by connecting nodes that (due to the overlapping introduced after projection) share the some points in common with other nodes.

After having traversed all the layers and units in order to get all the weights, a sample reduction is performed using KNN with a big number k of neighbours ~ 100 considering the distance from sample to its neighbours. Samples with the larges distances to their relatives (less dense neighbourhoods) are discarded. After that, the set is used to generate a lower dimensional projection of it using PCA. Despite having introduced UMAP as a good candidate for dimensionality reduction (and visualisation) the dimensionality of the data at certain given layers is too big to consider it (e. g: At layer 5b with kernel size 3, there are a total 96×384 channels resulting in a data set of more than 50K samples).

The mapper algorithm is used to apply a chosen clustering algorithm to the original data while using the samples from the projected data partitioned into 5 bins (or intervals) per dimension (having a total of 125 cubes on 3 dimensions) with a overlapping of just 10% between intervals. The chosen clustering algorithm for mapper to use is DBSCAN [22], since compared to other clustering methods provides a density based approach, that is ideal to the actual usage for this application (based on finding groups of samples in a reduced quadrant). DBSCAN has two parameters as input, being the maximum distance between samples and the minimum samples for a cluster to be considered.

All this mentioned steps can be found in the algorithm 6.

Algorithm 6: Channel-wise TDA for each layer

Input: neuronWeights, kernelSize, binsNumber, overlapping, cropPercentage, numberOfNeigbours

Output: Graph representing the channel-wise weights structure.

```
channels = []
for layer in neuronWeights do
    for neuron in layer do
        if neuron has kernelSize then
            for kernel in neuron do
                normalizedChannel = meanCenterScale(kernel)
                channels.append(flattenKernel(normalizedChannel))
    reducedSampleSet = KNNDensityCrop(cropPercentage, numberOfNeigbours)
    projectedData = PCA(reducedSampleSet, numComponents=3)
    graph = Mapper(reducedSampleSet, projectedData, binsNumber, overlapping)
    captions = createCaptions(graph)
    saveGraphVis(graph, captions)
```

Fine tuning the parameters

In order to fine-tune the value of the previous parameters (such as the overlapping, number of bins, minimum samples in a cluster and the maximum distance in a cluster) several things are taken into account:

- The resulting amount of edges and nodes. Having a denser graph with a lot of edges between neighbour nodes might be an indication that the overlapping used is too big. Having a high amount of nodes could be related to the amount of bins that have been set for every dimension.
- By having computed the kernel matrix of every channel as an image, we can compare the samples that fall into the same node. If the amount of samples is relatively big and their kernel matrices have a completely different pattern, the amount overlapping might be too high.
- If the partitions into the different bins have resulted in less samples than the minimum number considered in DBSCAN, no clusters will be computed, thus resulting in a biased graph with less points. Therefore, the number of minimum samples must be "tweaked" in order to consider smaller clusters.
- At the same time, the maximum distance between samples must be also fined-tuned in order not to avoid losing meaning-full connections between two similar samples.

Graph interpretation

The resulting graph is interpreted and saved as an interactive graph (based on the visualisation framework D3 [23]) as an HTML file. In order to help with the actual node repre-

sentation and characteristics, graph nodes are sized proportionally to the actual number of elements that they contained. At the same time, custom tool-tips are created for each node where the kernels (representing each sample inside a node) are saved as grey-scaled images in order to aim the interpretation.

With those previous measures taken into account, the final visualisation can be seen in figure 36.

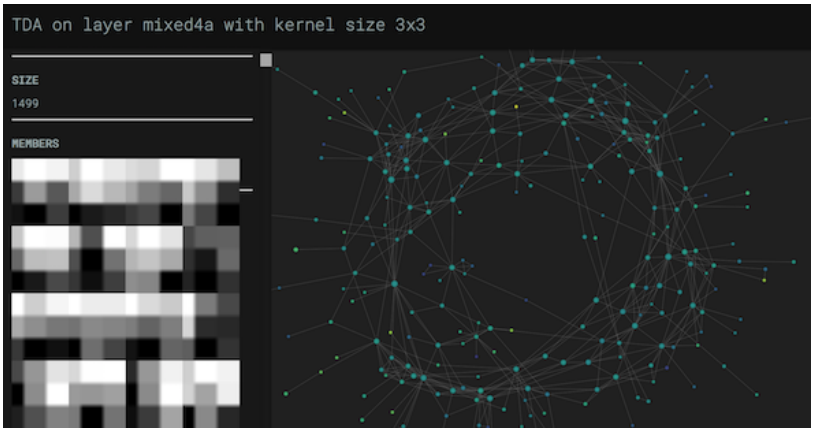
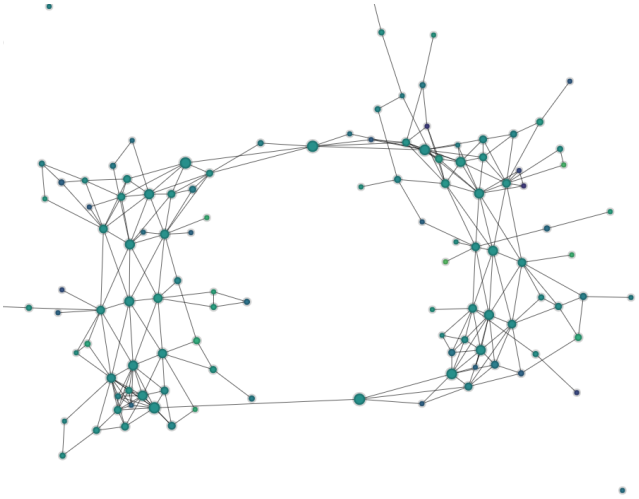


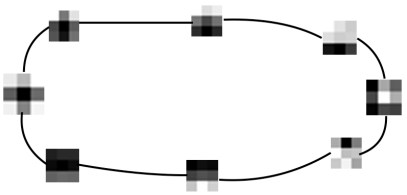
Figure 36: Final TDA visualisation on a layer

Once TDA is applied to every layer, the final visualisations (shown in figure 36) are simplified in order to improve their intrepretability for this document. That simplification can be understood as taking a kernel as representation for an entire node, thus providing more information on the characteristics of each node in the actual graph. Those simplifications can be seen in the following figure, corresponding to the resulting graph for the 3×3 unit kernels in layer *mixed5a*.

The same procedure is applied in the rest of the layers, which its results that can be found in section 7.3 the appendix.



(a) Topological representation of layer *mixed5a* with kernel size 3



(b) Simplified representation including kernel samples as nodes

Figure 37: Resulting graph after applying TDA on 3×3 kernel size channels in layer

The resulting graph for each layer leads to a structure that can be understood as an actual circle. By means of the simplification process shown in the previous paragraph, it can be seen (as the right image in figure 37 shows) that each kernel representing an entire node slightly differs from the actual adjacent nodes.

The findings from our previous results in this results clearly show a similar behaviour as the simple model found at the state of the art 2. These findings allow to extract several conclusions:

Having used kernels' channels for each layers as subject of study has helped us to get the general idea of their inner structure. Despite being grouped by neurons, layers in the actual architecture introduced should be considered as a collection of channels that try to mimic all the possible patterns feasible in a 3×3 or 5×5 image.

Despite being a complex architecture, we still can observe that in the left image in figure 37, there are less connected nodes in the centres compared to the actual sides. This behaviour could be considered not ideal since, in order to mimic an actual primary visual cortex, since all the possible patterns should result in a even dense connected graph at any position of the given ring. Nonetheless, we can argue that this behaviour could be tied to the actual input dataset and it would actually diverge if the dataset that is used to train the architecture is changed.

5 Conclusions

In this master thesis a research for finding behaviours in the components such as neurons of a given convolutional network have been carried out using three different approaches with mixed results.

On the supervised approach, a first exploration of the actual optimised input images allowed us to find potential patterns in the neurons. A further study and comparison with the correspondent neuron weights did help to diagnose how the actual patterns were represented on the neuron weights.

On the next step, implementing neuron classifiers based on some of the previous found patterns has proven to be useful in order to detect not which units are responsible of such behaviours but also the layers as well.

Despite somehow being able to partition the data into several clusters and extracting the averaged kernels that could represent them, the dimensional reduction of the unit weights into a 2-Dimensional embedded space did provide meaningful insights on the layer structure, showing a shapeless or circular structure.

Finally, a finer grade Topological Data Analysis on the actual layers have been incredibly helpful providing a global definition to be able to understand the layers.

All in all, we deem this study important in the path towards understanding the Convolutional Neural Network inner workings at, even, different levels that a CNN architecture can be divided in.

6 Future work

In this same research line, the different types of behaviours could be used in models that exploit the characteristics of those neurons. Foe example, border neurons behaviours could be useful to change image borders and creating simpler models by using unit weights with an already known behaviour.

The advantages and results that the usage of TDA has provided could be profitable for several research lines.

Firstly, after the idea of understanding layers as a group of channels with different patterns, CNN weights initialisation strategies could be studied in order to provide values prior training to the actual weights that allow for faster training times on different architectures.

Secondly, since knowing that the actual layers in an architecture would ideally converge to an actual ring structure, insights could be provided on how a given architecture is evolving by providing topological representations of its weights every certain number of times during training.

References

[1] Y. LeCun et al. "Gradient-Based Learning Applied to Document Recognition". In: *Proceedings of the IEEE* 86.11 (Nov. 1998), pp. 2278–2324.

[2] Christian Szegedy et al. "Going Deeper with Convolutions". In: *CoRR* abs/1409.4842 (2014). arXiv: 1409.4842. URL: http://arxiv.org/abs/1409.4842.

[3] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[5] Jason Yosinski et al. "Understanding Neural Networks Through Deep Visualization". In: *CoRR* abs/1506.06579 (2015). arXiv: 1506.06579. URL: http://arxiv.org/abs/1506.06579.

[6] Dumitru Erhan et al. "Visualizing Higher-Layer Features of a Deep Network". In: *Technical Report, Univeristé de Montréal* (Jan. 2009).

[7] Chris Olah, Alexander Mordvintsev, and Ludwig Schubert. "Feature Visualization". In: *Distill* (2017). https://distill.pub/2017/feature-visualization. DOI: 10.23915/distill.00007.

[8] Anh Mai Nguyen, Jason Yosinski, and Jeff Clune. "Multifaceted Feature Visualization: Uncovering the Different Types of Features Learned By Each Neuron in Deep Neural Networks". In: *CoRR* abs/1602.03616 (2016). arXiv: 1602.03616. URL: http://arxiv.org/abs/1602.03616.

[9] Aravindh Mahendran and Andrea Vedaldi. "Understanding Deep Image Representations by Inverting Them". In: *CoRR* abs/1412.0035 (2014). arXiv: 1412.0035. URL: http://arxiv.org/abs/1412.0035.

[10] Anh Mai Nguyen et al. "Synthesizing the preferred inputs for neurons in neural networks via deep generator networks". In: *CoRR* abs/1605.09304 (2016). arXiv: 1605.09304. URL: http://arxiv.org/abs/1605.09304.

[11] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. "Deep inside convolutional networks: Visualising image classification models and saliency maps". In: *arXiv preprint arXiv:1312.6034* (2013).

[12] Chris Olah et al. "The Building Blocks of Interpretability". In: *Distill* (2018). DOI: 10.23915/distill.00010. URL: https://distill.pub/2018/building-blocks.

[13] Fred Hohman et al. "Summit: Scaling Deep Learning Interpretability by Visualizing Activation and Attribution Summarizations". In: *CoRR* abs/1904.02323 (2019). arXiv: 1904.02323. URL: http://arxiv.org/abs/1904.02323.

[14] Rickard Brüel Gabrielsson and Gunnar E. Carlsson. "A look at the topology of convolutional neural networks". In: *CoRR* abs/1810.03234 (2018). arXiv: 1810.03234. URL: http://arxiv.org/abs/1810.03234.

[15] David H Hubel and Torsten N Wiesel. "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex". In: *The Journal of physiology* 160.1 (1962), pp. 106–154.

[16] *lucid: A collection of infrastructure and tools for research in neural network interpretability.* URL: https://github.com/tensorflow/lucid.

[17] Martin Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[18] Xiaoyong Yuan et al. "Adversarial Examples: Attacks and Defenses for Deep Learning". In: *CoRR* abs/1712.07107 (2017). arXiv: 1712.07107. URL: http://arxiv.org/abs/1712.07107.

[19] Leland McInnes et al. "UMAP: Uniform Manifold Approximation and Projection". In: *The Journal of Open Source Software* 3.29 (2018), p. 861.

[20] Nathaniel Saul and Hendrik Jacob Van Veen. *MLWave/kepler-mapper*. 2017. URL: https://github.com/scikit-tda/kepler-mapper.

[21] Gurjeet Singh, Facundo Mémoli, and Gunnar E Carlsson. "Topological methods for the analysis of high dimensional data sets and 3d object recognition." In: *SPBG*. 2007, pp. 91–100.

[22] Martin Ester et al. "A density-based algorithm for discovering clusters in large spatial databases with noise." In:

[23] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. "D^3 data-driven documents". In: *IEEE transactions on visualization and computer graphics* 17.12 (2011), pp. 2301–2309.

[24] *Inception V1 keras implementation.* URL: https://gist.github.com/joelouismarino/a2ede9ab3928f999575423b9887abd14%5C#file-googlenet-py.

[25] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep sparse rectifier neural networks". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. 2011, pp. 315–323.

7 Appendix

Input image optimisation parameters and regularisation

Lucid provides some regularisation measures to improve the quality of the optimised input image.

- Padding the input by 16 pixels (with value .5) to avoid edge artefacts.
- Jittering by up to 8 pixels.
- Scaling by a factor randomly selected from this list: [0.9, 0.92, 0.94, 0.96, 0.98, 1.0, 1.02, 1.04, 1.06, 1.08, 1.1]
- Rotating by an angle randomly selected from this list; in degrees ranging from -10 to +10.
- Jittering a second time by up to 4 pixels.

However, further regularisation (by moving to a color-decorrelated fourier-transformed space, available as an option in Lucid), with a total of 2048 iterations using Adam optimiser and a learning rate of 0.05.

7.1 Full GoogleNet Architecture

As a complementary measure (and in order to clarify any methodology decisions that could be related to the model architecture) the full GoogleNet architecture [25] is introduced.

The amount of units, output dimensions and kernel sizes can be seen in the following table:

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj
convolution	7×7/2	112×112×64	1						
max pool	3×3/2	56×56×64	0						
convolution	3×3/1	56×56×192	2		64	192			
max pool	3×3/2	28×28×192	0						
inception (3a)		28×28×256	2	64	96	128	16	32	32
inception (3b)		28×28×480	2	128	128	192	32	96	64
max pool	3×3/2	14×14×480	0						
inception (4a)		14×14×512	2	192	96	208	16	48	64
inception (4b)		14×14×512	2	160	112	224	24	64	64
inception (4c)		14×14×512	2	128	128	256	24	64	64
inception (4d)		14×14×528	2	112	144	288	32	64	64
inception (4e)		14×14×832	2	256	160	320	32	128	128
max pool	3×3/2	7×7×832	0						
inception (5a)		7×7×832	2	256	160	320	32	128	128
inception (5b)		7×7×1024	2	384	192	384	48	128	128

Figure 38: GoogleNet layer specifications

[25] High level implementation can be found in [24]

From an implementation point of view, its worth mentioning that all convolutions use a **1 pixel stride**, **same padding** which enforces the output feature space to be equal to the input space (filling the surroundings of the input space with 0's). Also, **ReLU** [25] is used as **activation function**.

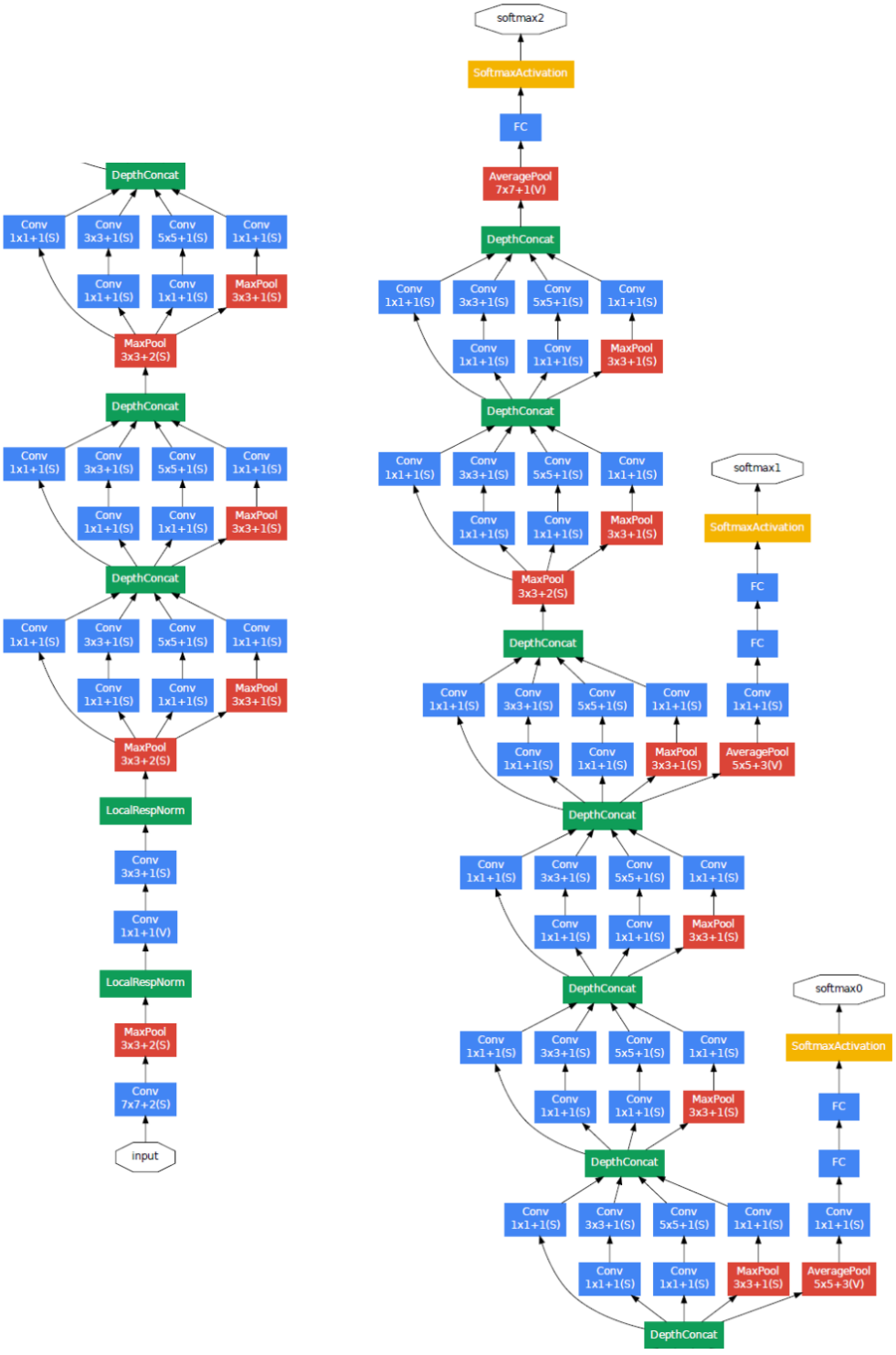


Figure 39: Full GoogleNet architecture

1×1 convolutional units

Being present in the chosen model (Googlenet), there are several 1×1 convolutional units that perform similar tasks with different objective.

Since the majority of CNN models increase the number of units on each layer, results in a high amount of output features (with high dimensionality) on the deeper layers of the network.

Having a higher amount of input channels compared to its output ones, those 1×1 convolutional units perform a channel wise dimensionality reduction. This could be also translated in less computational and storage memory, since the majority of the representativity of the features has been maintained whilst reducing the channel dimensionality. Those unit can be found prior a convolutional unit with a greater kernel size (e. g: 1×1 units prior a 3×3 or a 5×5 convolutional units in figure 39).

However, 1×1 units found inside an inception layer with no convolutional units after them (e.g after the 3×3 pooling units inside the inception layer), are used to create artificial features by a linear combination of each of them (by means of the convolutional unit weights).

Abstraction levels on the *mixed* layers

Once having extracted the optimised input images corresponding to each unit (inside each layer) a small amount of images of each layer are going to be displayed to understand the amount of complexity (or abstraction) that each layer has.

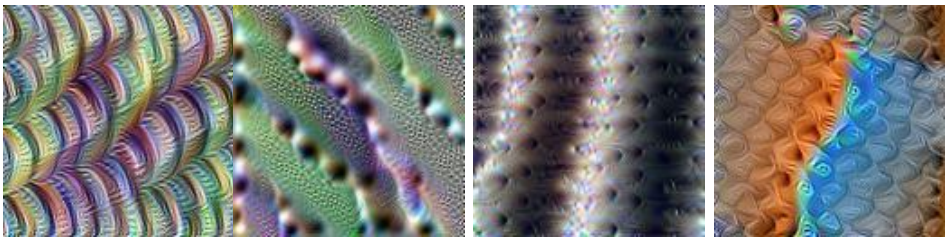


Figure 40: *mixed3a* input optimised images

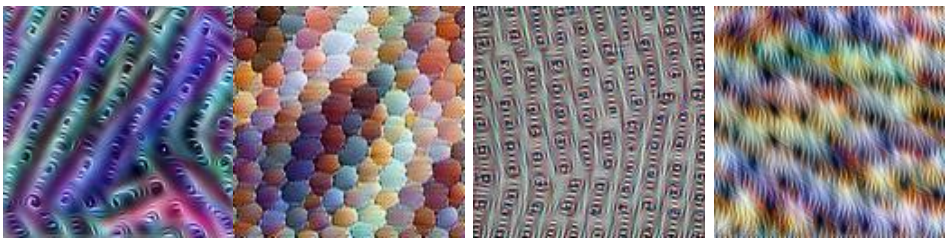


Figure 41: *mixed3b* input optimised images

Figure 42: *mixed4a* input optimised images

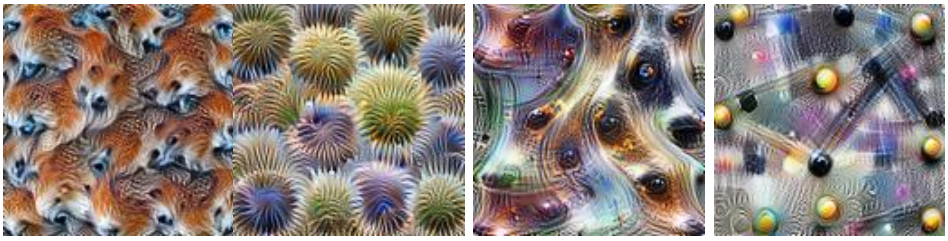


Figure 43: *mixed4b* input optimised images

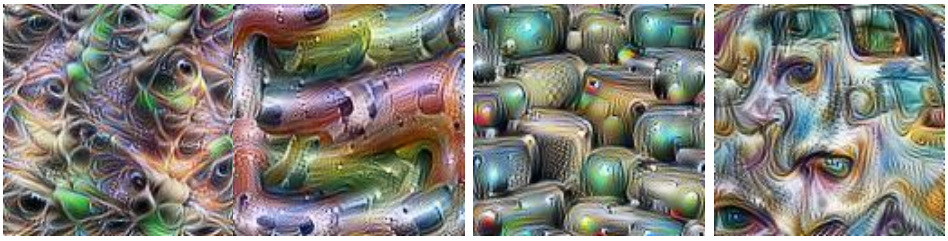


Figure 44: *mixed4c* input optimised images

Figure 45: *mixed4d* input optimised images

Figure 46: *mixed4e* input optimised images

Figure 47: *mixed5a* input optimised images

Figure 48: *mixed5b* input optimised images

Considering dimensionality reduction methods

Despite having other available dimensionality reduction methods for our clustering section, It has been decided to use UMAP to reduce the dimensions of the kernel matrices.

Some of the reasons to choose UMAP are:

- Is faster than other dimensionality reduction methods (t-SNE, MDS, etc.) since it is very efficient at embedding large high dimensional datasets. [26].
- Can be used for more than just visualisation, can be also used as a general purpose dimension reduction technique for any further methods.
- Often performs better at preserving aspects of the data structures than t-SNE. This means that it will usually provide a better "big picture" view of the data as well as keeping local neighbour relations, which suits UMAP for clustering over t-SNE.

Parameters used in cluster and TDA

mapper - TDA

- Projection function: PCA with 3 components.
- Clustering algorithm: DBSCAN with epsilon = 0.5 and min_samples = 3 for kernel size 3 and epsilon = 2 and min_samples = 3 for kernel size 5
- Overlapping: 10%

[26]Benchmarks against PCA and t-SNE can be found in `https://umap-learn.readthedocs.io/en/latest/benchmarking.html`

- Bins per dimension: 5

Hierarchical clustering

- Linkage method: Ward

Spectral clustering

- rbf kernel based afinity matrix.

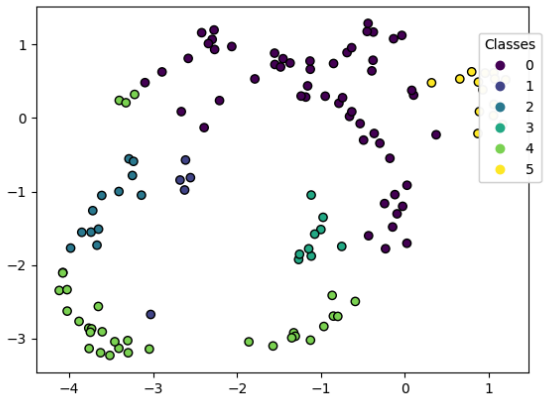
UMAP

- Number of neighbours to consider: 30
- Minimum distance between points: 0.0
- Number of output dimensions: 2

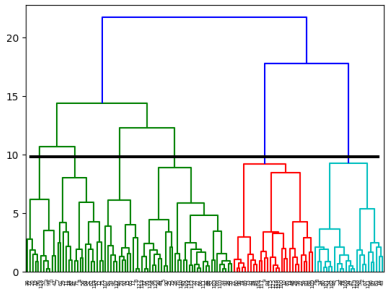
7.2 Complete results on unit-wise clustering

Layer *mixed3a*

After having dimensionally reduced units in layer *mixed3a*, and applied clustering to them, we find several feasible cluster cuts from which we decide to pick the one with 6 clusters (shown in the right-most hierarchical clustering dendrogram. On the left we can observe the partition of those six clusters in the lowered space.



(a) 2D embedding of average kernels in layer *mixed3a*



(b) Hierarchical clustering dendrogram for layer *mixed3a*

Figure 49: 2D embedding & Hierarchical clustering dendrogram of the average kernels with kernel size 3×3 in layer *mixed3a*

In order to compute the characteristics of each cluster, we focus on providing the a short explanation of kernels that each of the cluster elements are distinguished for (if and only if the cluster haves a meaning full number of elements compared to the rest).

Cluster num.	0	1	2	3	4	5
# units	55	5	12	9	33	14

From left to right in the previous table of cluster size, clusters in this layer can be identified by:

- Units that have an averaged kernel with a centred positive values with several different combinations of negative ones surrounding them.
- Units that have a diagonal shaped pattern of negative values.
- Units that have an averaged kernel with a centred negative value with positive ones surrounding them.
- Units that have a vertical pattern and/or an horizontal one in their top left part of the image based of negative values.
- Units that have combinations of negative values with around a negative centre on 2 of their 3 rows.
- Units that have a vertical pattern and/or an horizontal one in their bottom right part of the image based of negative values.

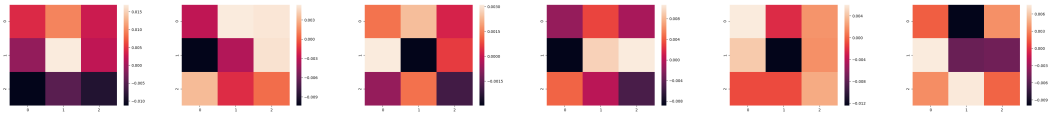


Figure 50: *mixed3a* 5×5 kernels that represent each cluster.r

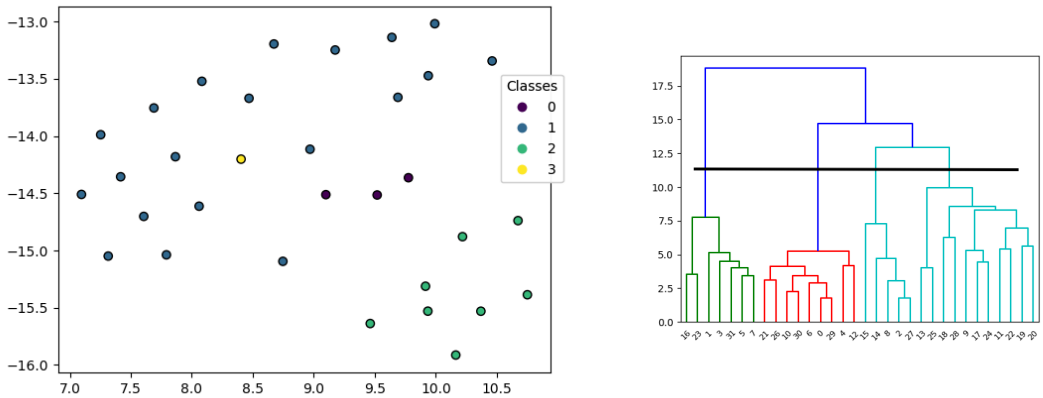


Figure 51: 2D embedding & Hierarchical clustering dendrogram of the average kernels with kernel size 5×5 in layer *mixed3a*

Cluster num.	0	1	2	3
# units	3	20	8	1

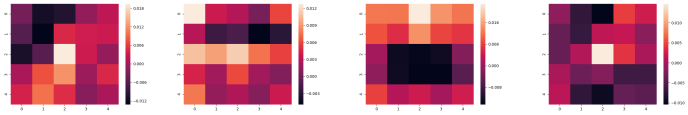


Figure 52: *mixed3a* 5×5 that represent each cluster.

The clustering of 5×5 units provide the following clustered unit behaviour (based on their averaged kernel):

- The first cluster is formed by units distinguishable by a kernel that provides a negative pattern at the upper part of the kernel.
- The second the biggest cluster, units in this first cluster provide a negative square shaped pattern around the centre of the kernel.
- Third cluster is found by units that show a behaviour based on their pattern (negative values) found in the lower part of the kernel.
- The last (and fourth) cluster, only consists on one unit that haves a pattern based on negative values on its upper left kernel and on all the lower part of it.

Layer *mixed3b*

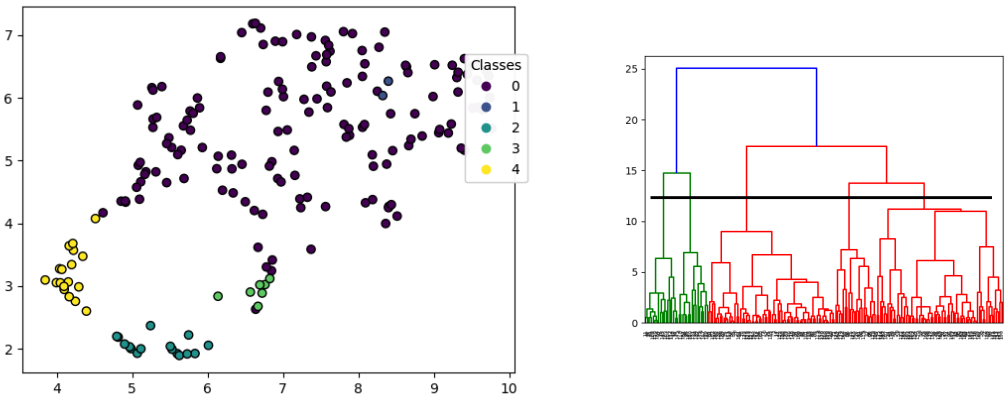


Figure 53: 2D embedding & Hierarchical clustering dendrogram of the average kernels with kernel size 3×3 in layer *mixed3b*

Cluster num.	0	1	2	3	4
# units	149	2	16	7	18

Having a big cluster, the units with kernel size 3×3 can be characterised by their pattern (identifiable by having central a negative value and a combination of negative and positive values on their surroundings seen in in the first image on figure 54).

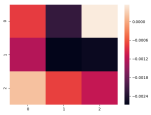
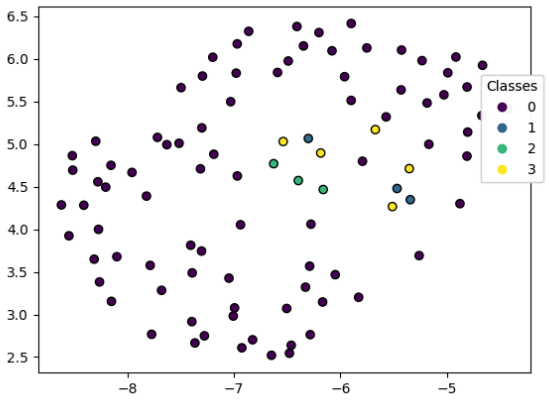
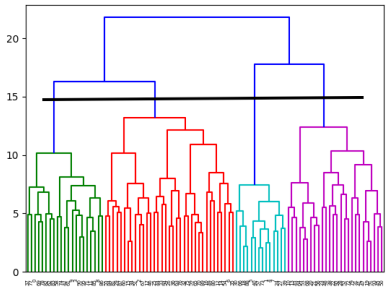


Figure 54: *mixed3b* 3×3 kernels that represent each cluster.



(a) 2D embedding of average kernels in layer *mixed3b*



(b) Hierarchical clustering dendrogram for layer *mixed3b*

Figure 55: 2D embedding & Hierarchical clustering dendrogram of the average kernels with kernel size 5×5 in layer *mixed3b*

Cluster num.	0	1	2	3
# units	85	3	3	5

Due to the difference in cluster sizes (85 of the first cluster versus the rest), no apparent pattern is found upon inspection due to the diversity of the actual kernels.

Layer *mixed4a*

From the 3×3 kernel units in layer *mixed4a*, two big clusters are considered to be important.

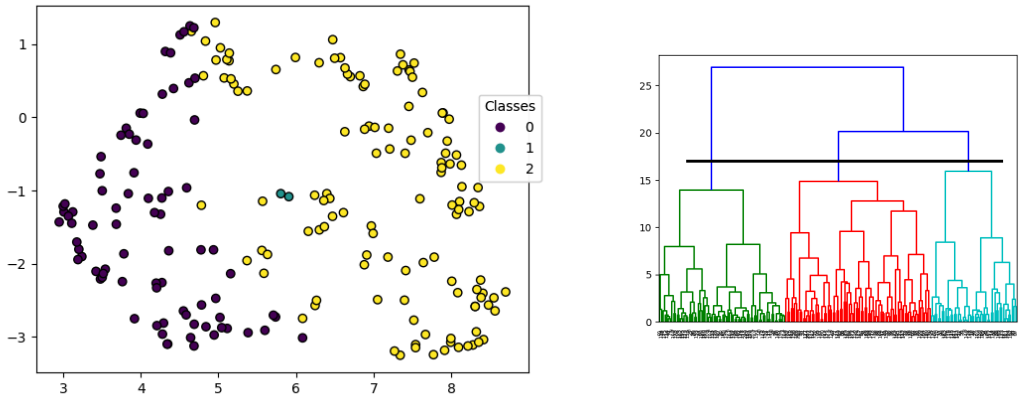


Figure 56: 2D embedding & Hierarchical clustering dendrogram of the average kernels with kernel size 3×3 in layer *mixed4a*

Cluster num.	0	1	2
# units	80	2	122

The main difference between them is the central value in their kernels. For the first cluster, the vast majority of the 80 elements contained have a positive value on their central kernel position. Being the other way around, all of the 122 elements for the third cluster have a negative value on the same position. Representations for both of them can be seen in fig. 57

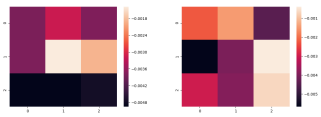
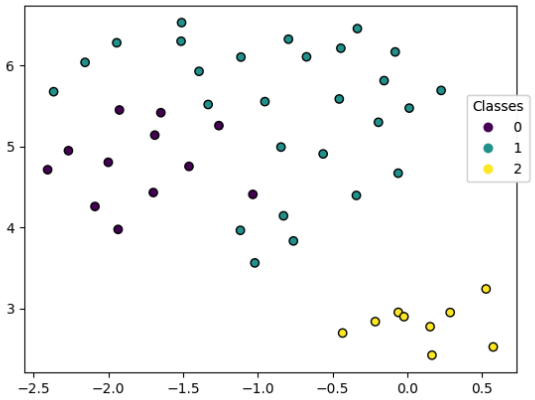
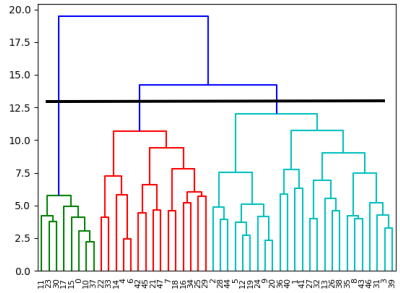


Figure 57: *mixed4a* 3×3 kernels that represent the first and third clusters





(a) *mixed4a*

Figure 58: 2D embedding & Hierarchical clustering dendrogram of the average kernels in layer *mixed4a*

Cluster num.	0	1	2
# units	12	27	9

From the 3 clusters found for the units in layer *mixed4a* kernel size, only the third cluster (being the most distant in the previous embedded representation in fig. 56) can be characterised by having a horizontal and squared pattern of negative values in the centre.

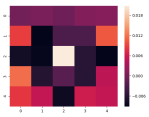
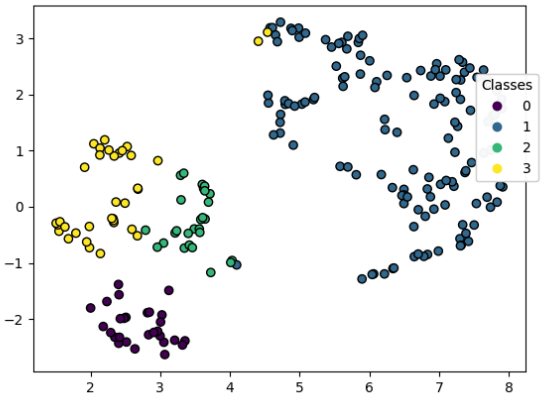
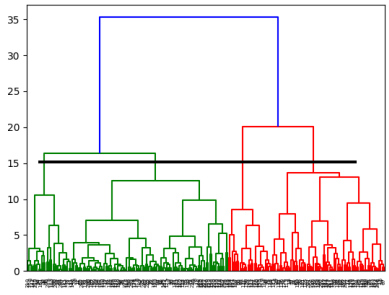


Figure 59: *mixed4a* 5×5 kernel representing the third cluster

Layer *mixed4b*



(a) 2D embedding of average kernels in layer *mixed4b*



(b) Hierarchical clustering dendrogram for layer *mixed4b*

From layer *mixed4b*, four clusters have been found on units with kernel size 3×3 and are sized as shown in table 6.

Cluster num.	0	1	2	3
# units	28	134	27	35

Table 6: Size of each cluster in layer *mixed4b* for units with kernel size 3×3

Those four clusters can be respectively characterised by:

- Negative values on both left and right sides of the central row in their kernel or in the last row.
- Being the biggest cluster, units in the second cluster are characterised bye having positive values in their kernel centre and a negative values on their corners.
- Units in the third cluster that have a positive value in the centre and negative values in some of his adjacent neighbours (top-right-bottom or left positions).
- Units in the fourth cluster do have a negative values for their central row and / or the first row of their kernels.

As similarly done with the cluster size, unit kernels that respectively represent each one of the explained clusters can be seen at figure 61.

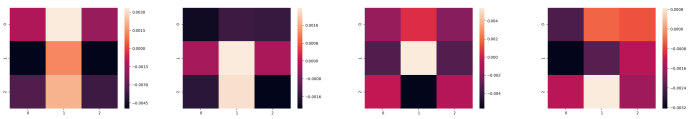


Figure 61: *mixed4b* 3×3 kernels representing each cluster

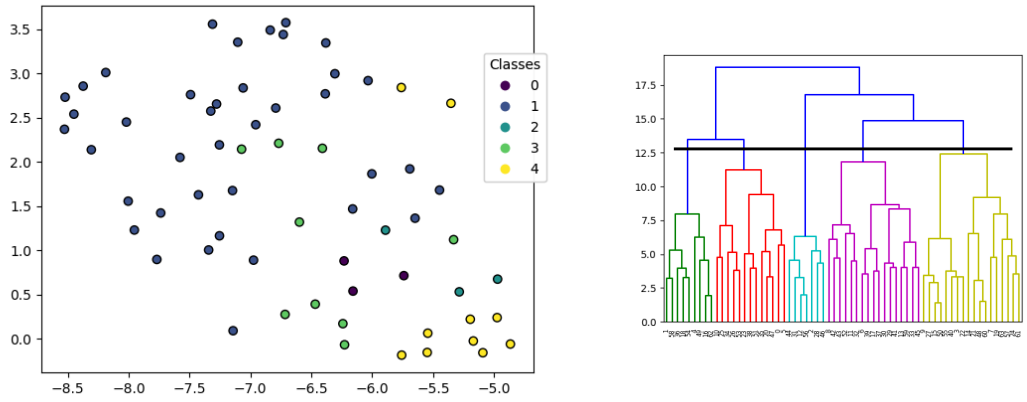


Figure 62: 2D embedding & Hierarchical clustering dendrogram of the average kernels in layer *mixed4b*

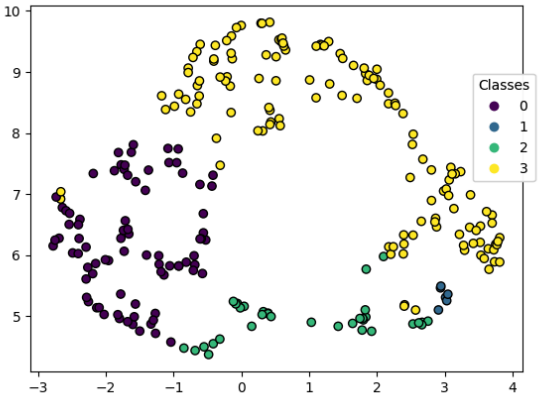
For the 5×5 kernel sized units in layer *mixed4b*, five clusters are found with the following sizes (in table 7).

Cluster num.	0	1	2	3	4
# units	3	39	3	9	10

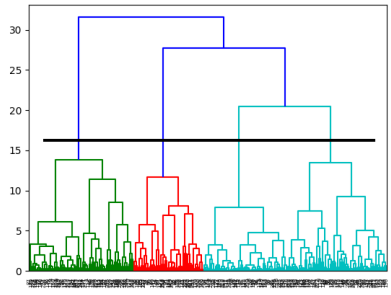
Table 7: Size of each cluster in layer *mixed4b* for units with kernel size 5×5

Unfortunately, no pattern has been found that could lead to an actual characterisation of clusters. This goes in accordance with the impure separation of clusters seen in figure 62.

Layer *mixed4c*



(a) 2D embedding of average kernels in layer *mixed4c*



(b) Hierarchical clustering dendrogram for layer *mixed4c*

Figure 63: 2D embedding & Hierarchical clustering dendrogram of the average kernels in layer *mixed4c*

By having partitioned embedded space of the 3×3 kernel sized units in layer *mixed4c* in four clusters (shown in fig. 63), we have obtained the following cluster sizes (as table 8 shows):

Cluster num.	0	1	2	3
# units	83	6	34	133

Table 8: Size of each cluster in layer *mixed4c* for units with kernel size 3×3

Having two major clusters, those two are characterised respectively by:

- Having a positive value on the kernel centre and negative on their top row.
- A positive value in the kernel centre and negative ones on the bottom row (or even a the four corners).

Representative unit kernels for each of the biggest clusters can be seen at fig. 64.

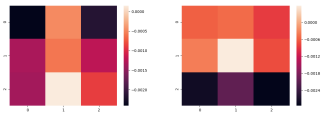


Figure 64: *mixed4c* 3×3 kernels that represent each cluster

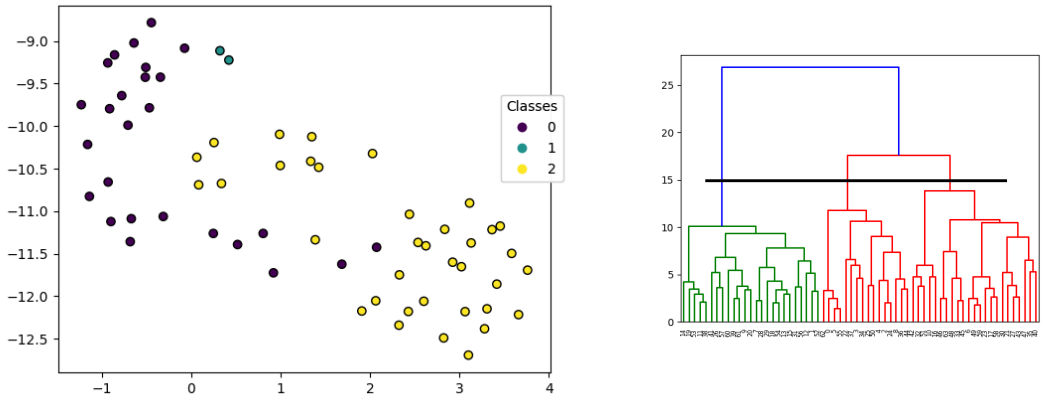


Figure 65: 2D embedding & Hierarchical clustering dendrogram of the average kernels with kernel size 5×5 in layer *mixed4c*

By having partitioned embedded space of the 3×3 kernel sized units in layer *mixed4c* in four clusters (shown in fig. 65), we have obtained the following cluster sizes (as table 9 shows):

Cluster num.	0	1	2
# units	26	2	36

Table 9: Size of each cluster in layer *mixed4c* for units with kernel size 5×5

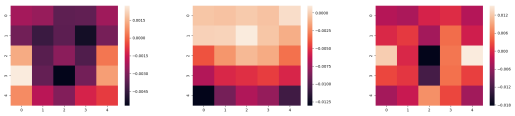


Figure 66: *mixed4c* 5×5 kernels from layer that are part of a cluster

Layer *mixed4d*

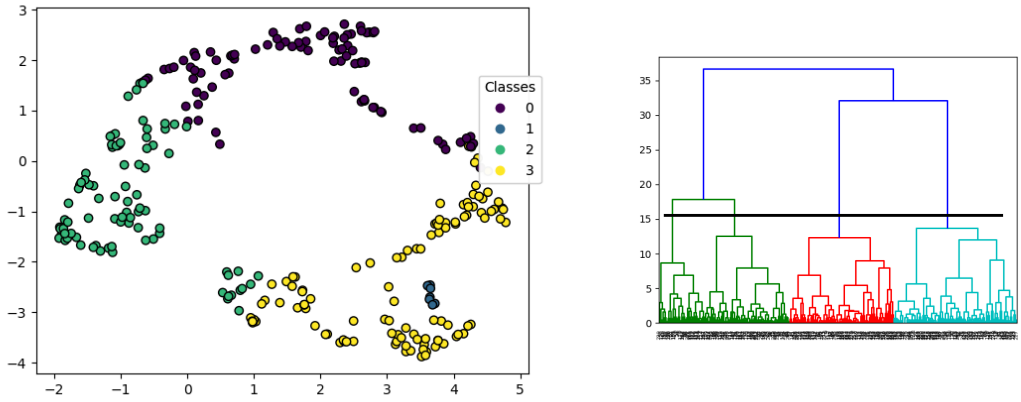


Figure 67: 2D embedding & Hierarchical clustering dendrogram of the average kernels with kernel size 3×3 in layer *mixed4d*

By having partitioned embedded space of the 3×3 kernel sized units in layer *mixed4d* in four clusters (shown in fig. 67), we have obtained the following cluster sizes (as table 10 shows):

Cluster num.	0	1	2	3
# units	89	7	86	106

Table 10: Size of each cluster in layer *mixed4d* for units with kernel size 3×3

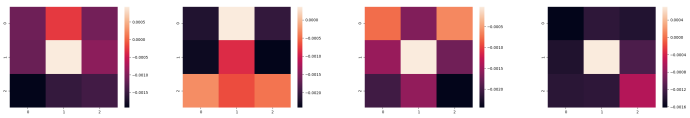
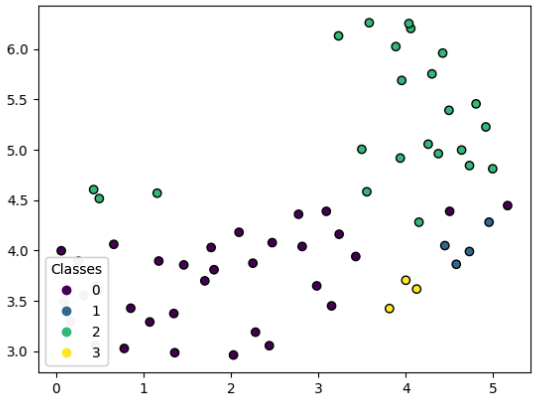
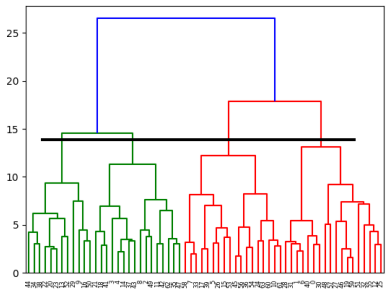


Figure 68: *mixed4d* 3×3 kernels that represent each cluster.



(a) 2D embedding of average kernels in layer *mixed4d*



(b) Hierarchical clustering dendrogram for layer *mixed4d*

Cluster num.	0	1	2	3
# units	34	4	23	3

Table 11: Size of each cluster in layer *mixed4d* for units with kernel size 5×5

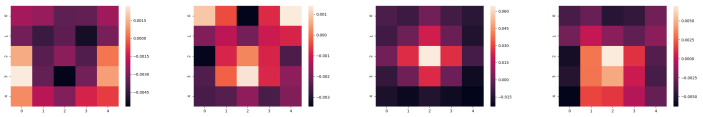
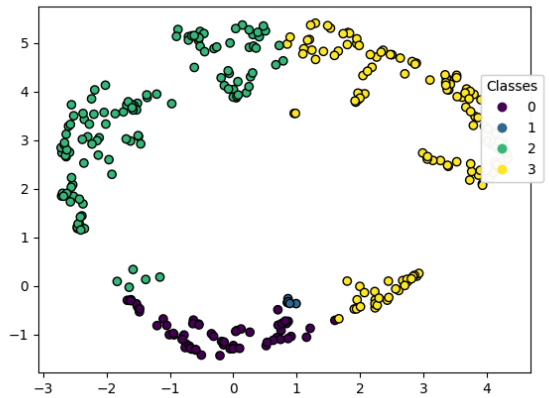


Figure 70: *mixed4d* 5×5 kernels that represent each cluster.

Layer *mixed4e*



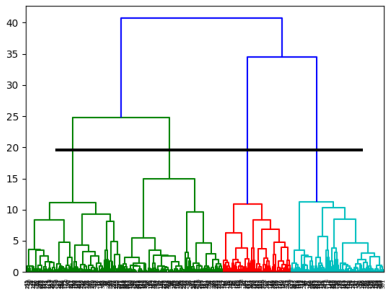


Figure 71: 2D embedding & Hierarchical clustering dendrogram of the average kernels with kernel size 3×3 in layer *mixed4e*

Cluster num.	0	1	2	3
# units	58	5	123	134

Table 12: Size of each cluster in layer *mixed4e* for units with kernel size 3×3

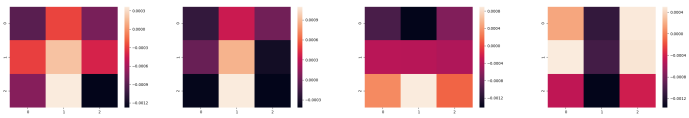


Figure 72: *mixed4e* 3×3 kernels that represent each cluster.

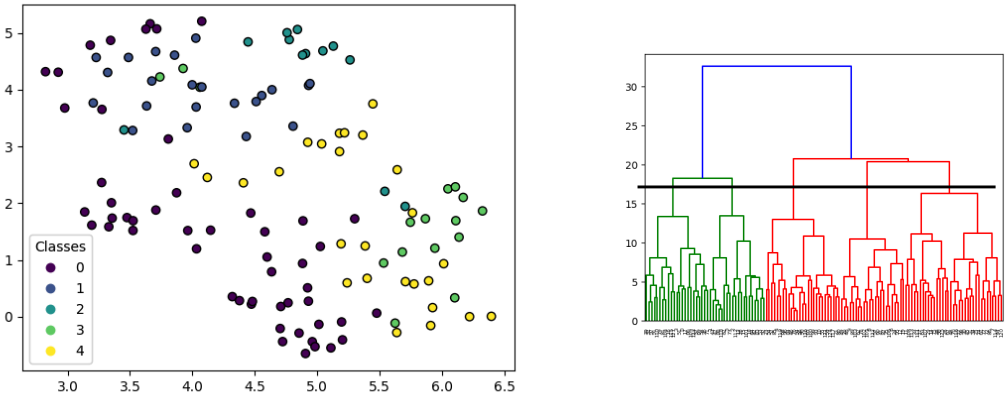


Figure 73: 2D embedding & Hierarchical clustering dendrogram of the average kernels with kernel size 5×5 in layer *mixed4e*

Cluster num.	0	1	2	3	4
# units	52	22	19	12	23

Table 13: Size of each cluster in layer *mixed4e* for units with kernel size 5×5

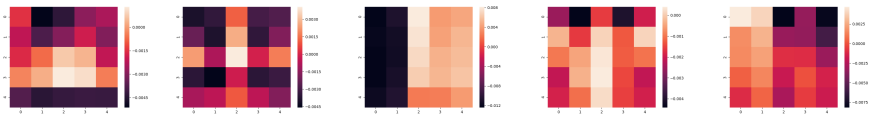


Figure 74: *mixed4e* 5×5 kernels that represent each cluster.

Layer *mixed5a*

3×3 kernel units in layer 5a have been separated into four clusters, shown at the left image in figure 75.

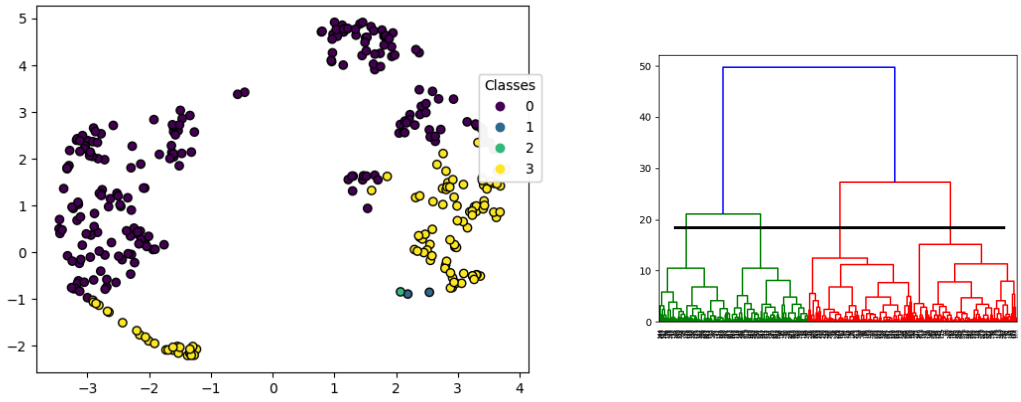


Figure 75: 2D embedding & Hierarchical clustering dendrogram of the average kernels with kernel size 3×3 in layer *mixed5a*

However, only 2 of them are actually representative of the actual layer, based on their sizes (found in table 14).

Cluster num.	0	1	2	3
# units	210	2	1	107

Table 14: Size of each cluster in layer *mixed5a* for units with kernel size 3×3

The actual difference between those clusters is located in the central kernel value. For the first cluster, that value remains positive. For the fourth one, it takes negative values.

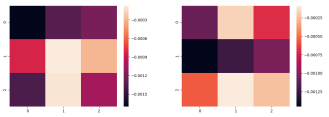


Figure 76: *mixed5a* 3×3 kernels that represent each cluster.

5×5 kernel units in layer 5a have been separated into three clusters, shown at the left image in figure 77.

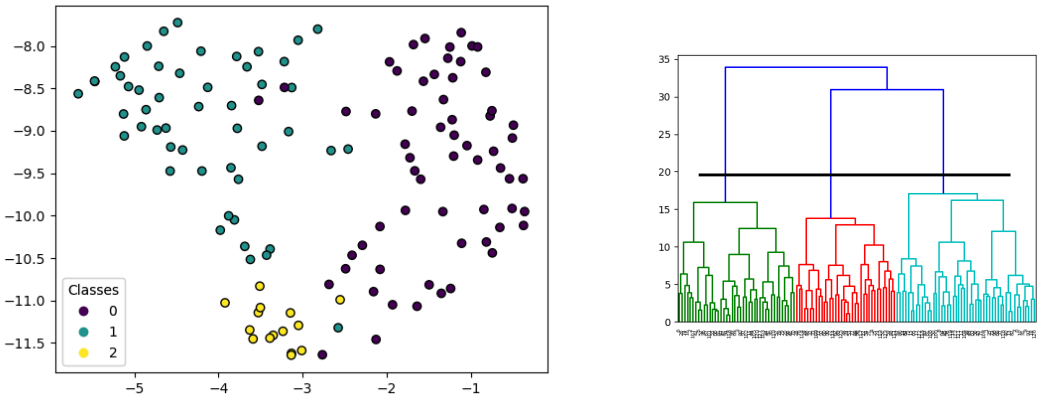


Figure 77: 2D embedding & Hierarchical clustering dendrogram of the average kernels with kernel size 5×5 in layer *mixed5a*

Cluster num.	0	1	2
# units	51	62	15

Table 15: Size of each cluster in layer *mixed5a* for units with kernel size 5×5

Having sizes as specified in table 15 and despite having a somewhat clear separation in figure 77, cluster characteristics have not been able to be obtained since each of the three clusters share a lot of similar kernels from the rest.

Layer *mixed5b*

3×3 kernel units in layer 5b have been separated into four clusters, shown at the left image in figure 78.

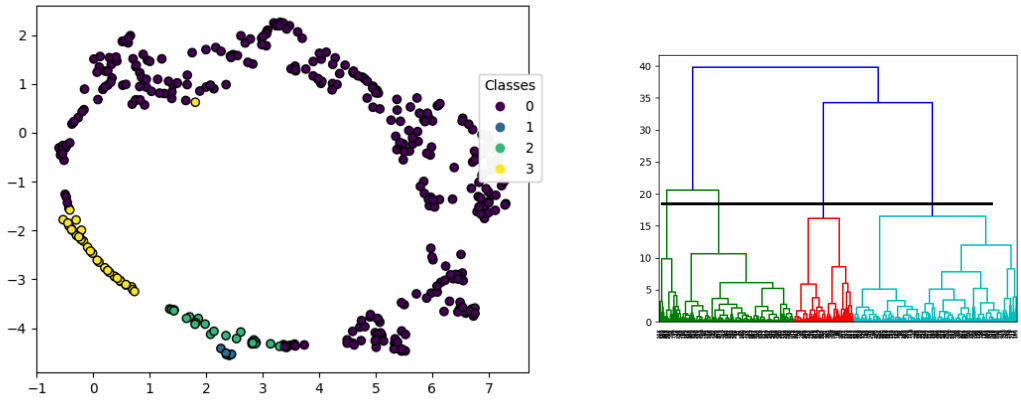


Figure 78: 2D embedding & Hierarchical clustering dendrogram of the average kernels with kernel size 3×3 in layer *mixed5b*

Cluster num.	0	1	2	3
# units	317	6	27	34

Table 16: Size of each cluster in layer *mixed5b* for units with kernel size 3×3

Having a big cluster (as shown in table 16) we can extrapolate that cluster as the actual layer behaviour, which has respectively the following characteristics and representation (observable in 79):

- Units corresponding to the first and second cluster do have a negative value in the centre position on the kernel and have up to two rows of negative values.

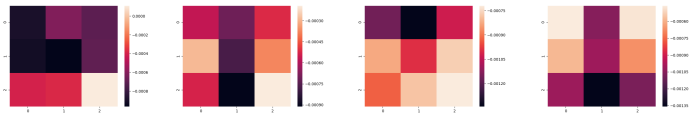


Figure 79: *mixed5b* 3×3 kernels that represent each cluster.

5×5 kernel units in layer 5b have been separated into three clusters, shown at the left image in figure 80.

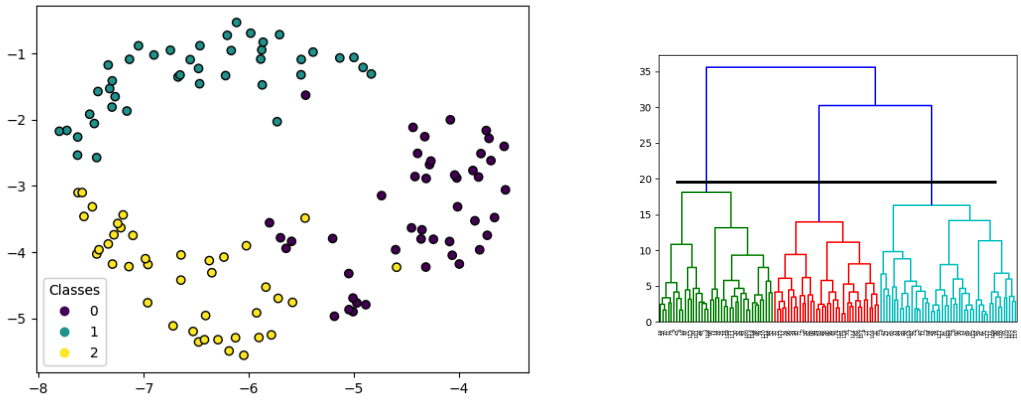


Figure 80: 2D embedding & Hierarchical clustering dendrogram of the average kernels with kernel size 5×5 in layer *mixed5b*

Cluster num.	0	1	2
# units	46	40	42

Table 17: Size of each cluster in layer *mixed5b* for units with kernel size 5×5

Having similar sizes (as shown in table 17) each cluster has respectively the following characteristics and representation (observable in 81):

- Units corresponding to the first and second cluster do have a negative value in the centre position on the kernel and have negative values can be found the central row or to and actual border (top or bottom).
- Units corresponding to the third cluster only have horizontal patterns (with negative values) found in the bottom or top row.

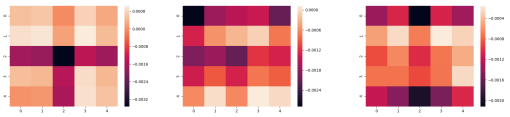
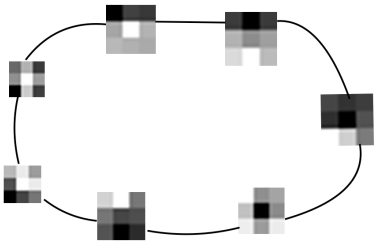


Figure 81: *mixed5b* 5×5 kernels, each representing a cluster

7.3 Complete results on TDA

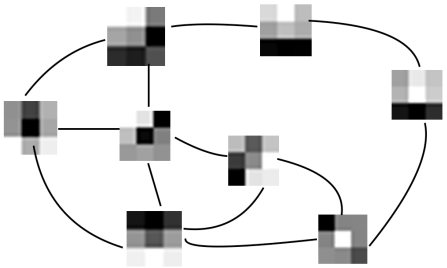
(a) Topological representation of layer *mixed3a* with kernel size 3



(b) Simplified representation including kernel samples as nodes

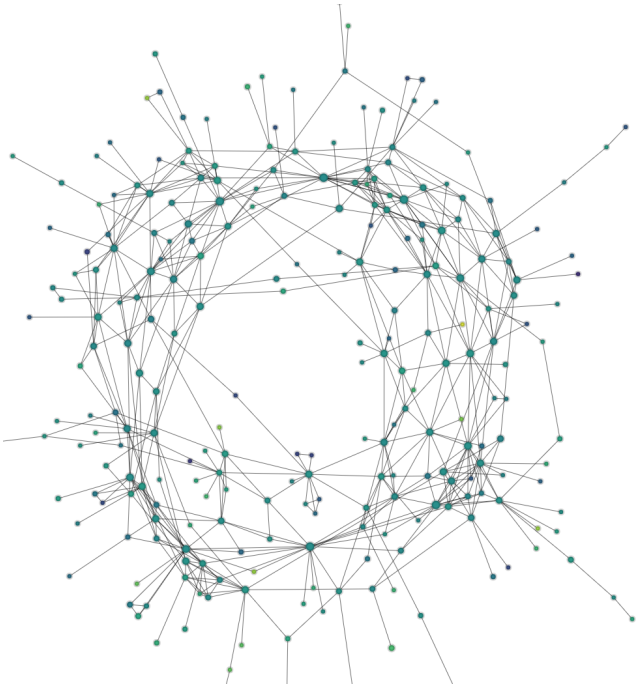
Layer *mixed3a*

(a) Topological representation of layer *mixed3b* with kernel size 3

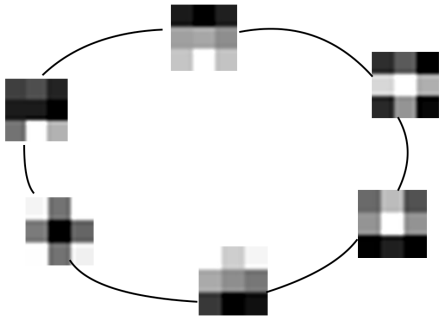


(b) Simplified representation including kernel samples as nodes

Layer *mixed3b*

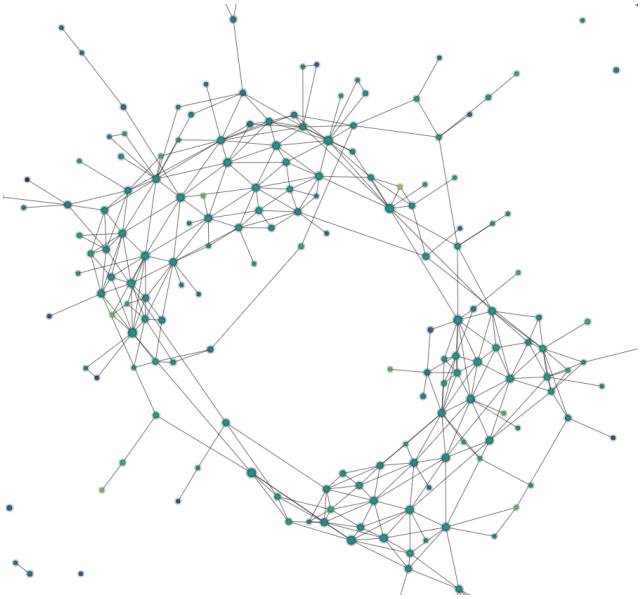


(a) Topological representation of layer *mixed4a* with kernel size 3

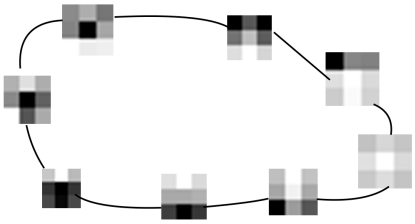


(b) Simplified representation including kernel samples as nodes

Layer *mixed4a*

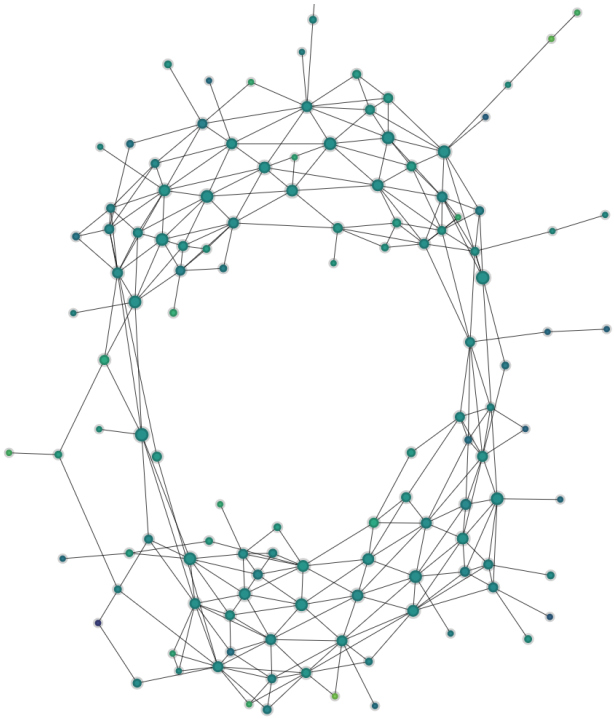


(a) Topological representation of layer *mixed4b* with kernel size 3

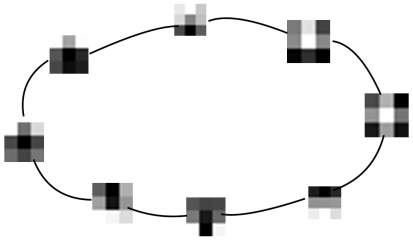


(b) Simplified representation including kernel samples as nodes

Layer *mixed4b*

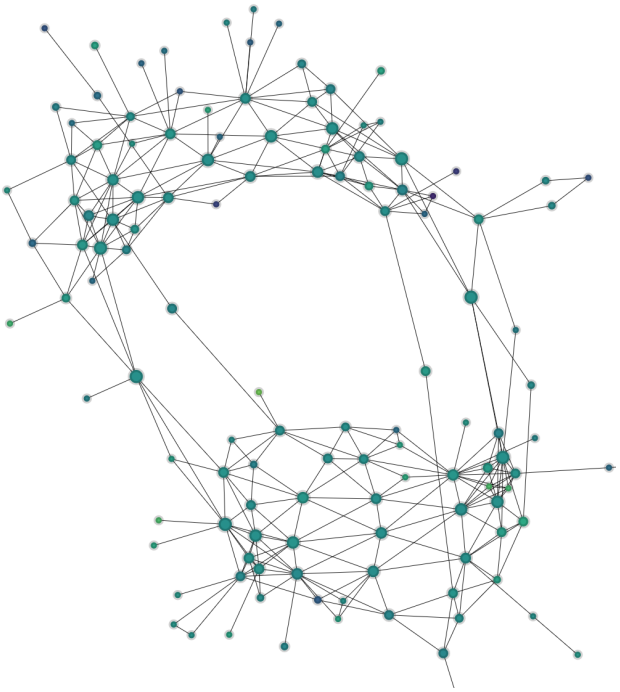


(a) Topological representation of layer *mixed4c* with kernel size 3

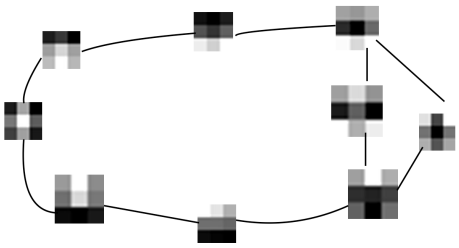


(b) Simplified representation including kernel samples as nodes

Layer *mixed4c*



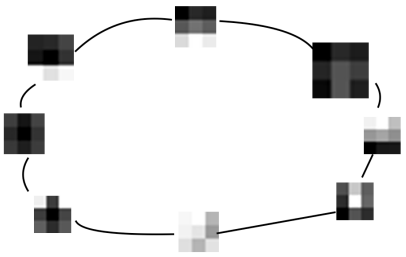
(a) Topological representation of layer *mixed4d* with kernel size 3



(b) Simplified representation including kernel samples as nodes

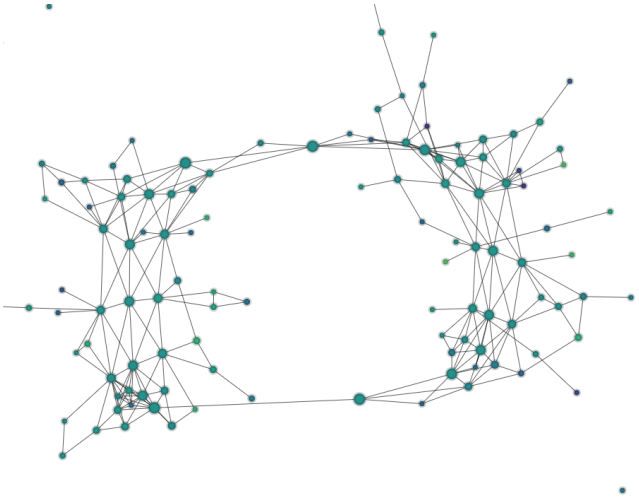
Layer *mixed4d*

(a) Topological representation of layer *mixed4e* with kernel size 3

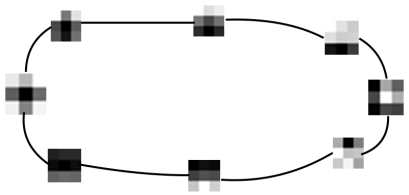


(b) Simplified representation including kernel samples as nodes

Layer *mixed4e*



(a) Topological representation of layer *mixed5a* with kernel size 3

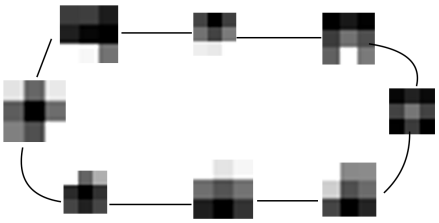


(b) Simplified representation including kernel samples as nodes

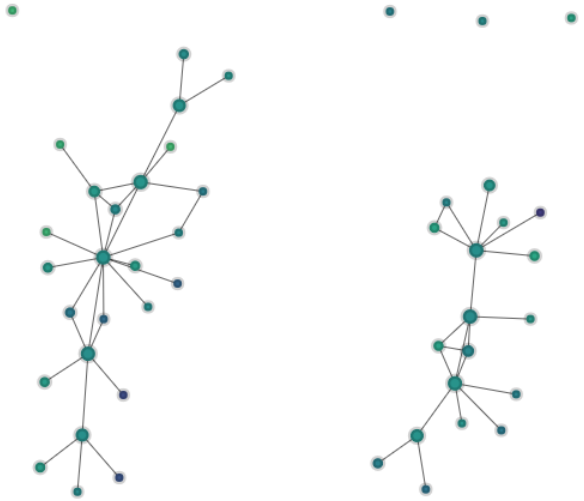
Figure 90: Topological representation of layer *mixed5a* with kernel size 5

Layer *mixed5a*

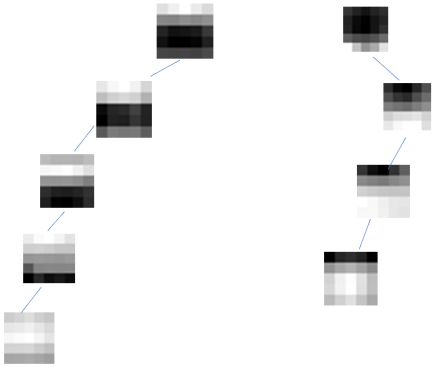
(a) Topological representation of layer *mixed5b* with kernel size 3



(b) Simplified representation including kernel samples as nodes



(a) Topological representation of layer *mixed5b* with kernel size 5



(b) Simplified representation including kernel samples as nodes

Layer *mixed5b*

Glossary

kernel In a convolutional neural network, weights store matrices used to convolve on the input features (also used in image processing). For each input channel, those matrices are called kernels. 2

lucid Feature visualisation and CNN interpretability package for python. Uses tensorflow as the deep learning framework.. 13, 14

matplotlib Visualisation package for python.. 13

numpy Scientific computing package for python.. 13

opencv Computer vision related package for python.. 26

protobuf Serialisation format used by Google.. 13

scikit-learn Machine learning package for python.. 13

seaborn Visualisation package for python.. 13

tensorflow Deep learning framework.. 13

unit Entity that belongs to a given layer and applies a operation (such a convolution) on a set of inputs (or channels). Also known as neuron. . 7

weights Links between different model layers. Represent the strenght between them (a.k.a filters).. 13

Acronyms

a.k.a also known as. 2

CNN Convolutional Neural Network. , ii, 1, 2, 3, 6, 9, 11, 14, 37, 38, v

ILSVRC ImageNet Large Scale Visual Recognition Competition. 7

KNN K-Nearest Neighbours. 33

MLP Multilayer Perceptron. 5

NN Neural Network. 9, 16

PCA Principal Component Analysis. 33, vii

ReLU Rectified Linear Unit. 3

SGD Stochastic gradient descent. 7

Std Standard deviation. 26

TDA Topological Data Analysis. , ii, iv, 11, 13, 26, 32, 33, 35, 37, 38, vii, xxiv

UMAP Uniform Manifold Approximation and Projection. 29, 33, viii

w.r.t with regards to. 4, 7